Objectbase Design: A Heuristic Approach.

DISSERTATION

DER WIRTSCHAFTSWISSENSCHAFTLICHEN
FAKULTÄT
DER UNIVERSITÄT ZÜRICH

zur Erlangung der Würde
eines Doktors der Informatik

vorgelegt von

THOMAS GROTEHEN
aus
Deutschland

genehmigt auf Antrag von

PROF. DR. KLAUS R. DITTRICH
PROF. DR. MARTIN GLINZ

Juli 2001

Die Wirtschaftswissenschaftliche Fakultät, Abteilung Informatik, gestattet hierdurch die Drucklegung der vorliegenden Dissertation, ohne damit zu den darin ausgesprochenen Anschauungen Stellung zu nehmen.

Zürich, den 16. April 2001

Der Lehrbereichsvorsteher: Prof Dr. Martin Glinz

When we see the pattern of the ripples in a pond, we know that this pattern is simply an equilibrium with the forces which exist: without any mental interference which is clouding them. And, when we succeed, finally, in seeing so deep into a man made pattern, that it is no longer clouded by opinions or by images, then we have discovered a piece of nature as valid, as eternal, as the ripples in the surface of a pond.
Christopher Alexander, The Timeless Way of Building, 1997

For Anke, Taya and Buma

Acknowledgements

This thesis is a result of the cooperation between the Computer Science Department of the University of Zurich and the Research and Architecture Groups of Swiss Bank Cooperation, Systor AG and CustomerCare AG.

I thank my supervisor Prof. Klaus R. Dittrich not only for his guidance but also for setting up this challenging yet rewarding type of close industry cooperation. On the industry side, I thank Rene Schwarb for being my mentor in many most interesting projects and continuously supporting my research.

I gratefully acknowledge the contributions of Prof. Martin Glinz who helped me improving this thesis by providing many valuable comments in his reviews.

I am particularly thankful for the cooperation with the members of the database group (Stella Gatziu, Barbara Rieche, Silvia Nittel, Dirk Jonscher, Stefan Scherrer, Dimitrios Tombros, Andreas Geppert, Andreas Behm, Hans Fritschi, Norbert Rump, Markus Kradolfer, and Thomas Meyer) and the team members in the industry projects (Regula Nebel, Roland Kannappel, Bernhard Bichmann, Arthur Neudeck, Niklaus Ruess, Roman Bargezi, Daniel Lipp and Christoph Sutter). Most of them have contributed to this thesis in one way or another.

I am indebted to Arthur Neudeck, Stefan Pluess and Silvano Maffeis for providing their Software for my evaluation.

I am especially thankful to Regula Nebel for eliminating many stylistic flaws in earlier versions of this thesis.

Last but not least, I would like to thank my family Anke, Taya and Buma. Without their loving support, all challenges in my life would be much harder to accomplish. Therefore, this thesis is dedicated to them.

Abstract

The interest in software design methodologies has been growing at a considerable rate in recent years. Nowadays, object-oriented design methodologies lack support for designing *objectbases*, e.g. shared object-oriented databases. This thesis presents a methodology extension called MeTHOOD[1] that supports designers in finding the road to *better designs* for these important components of many systems. MeTHOOD combines *measures* describing the objective, *heuristics* showing directions like a compass and *transformation rules* representing more concrete decision support (like roadsigns). It also provides a design tool that implements the extensions.

MeTHOOD makes design knowledge for conceptual *objectbase schemas* (conceptual object-oriented class schemas) more *tangible*. Although a large amount of this important knowledge is available in the literature (for example (Wirfs-Brock, Wilkerson et al. 1990; Lieberherr 1992; Henderson-Sellers, Edwards 1994; Graham 1995)), it is hardly usable for designers because it is very scattered, on different levels of abstraction and not integrated. Thus, the objective of this thesis is to *enable a design support process for efficient and continuous quality inspection and improvement of conceptual class schemas by efficiently providing design knowledge*. It is not necessary to re-invent a complete design methodology (including object model, high level process and notation) in order to meet this objective. Based purely on well-known object modeling languages (Booch, Rumbaugh et al. 1996; Firesmith, Henderson-Sellers et al. 1998) MeTHOOD provides an *extension* to existing methodologies. The core of MeTHOOD is a catalogue of design knowledge consisting of already existing as well as new object-oriented design heuristics, transformation rules and measures.

Design heuristics, measures and transformation rules are more or less known concepts. The core idea of this thesis is to *formalize* and to *integrate* them. This integration is called *MeTHOOD integration*. It makes implicit relationships between these concepts visible and benefits from them. It helps making "where"-decisions (designers are notified of problematic schema locations found by heuristics) and "how"-decisions (designers receive proposals for concrete schema improvements by transformation rules)

[1] **M**easures, **T**ransformation Rules and **H**euristics for **O**bject-**O**riented **D**esign

during design. In many cases, design alternatives can be generated. Deciding between design alternatives is simplified by continuous measurement of some of their interesting properties. Continuous measurement also helps selecting between different conflicting heuristics (one of the main problems of pure heuristics approaches like (Riel 1996)) and between competing transformation rules (resulting in different possible design alternatives).

The MeTHOOD integration is the base for the *iterative MeTHOOD design process*. It consists of the main activities *measurement*, *design inspection* and *design transformation*. These activities are supported by concrete measures and heuristics acting as a driver for transformations.

The process is supported by a design support system called *MEx*[2]. MEx provides semi-automatic *design monitoring*, *heuristic checking* and *design transformation* using the design knowledge from the MeTHOOD catalogue. It is an addition to a commercially available object-oriented design system with schema drawing facilities and generators/readers for different programming languages.

[2] MeTHOOD Expert

Table of Contents

Index of Figures

Index of Tables

Preface: The Dhigufinolhu design problem

Consider the following problem: a hotel chain has bought three small, disjoined islands on the Maledives and plans to connect these islands by simple wooden bridges. The design task is to provide the most economical solution for these bridges.

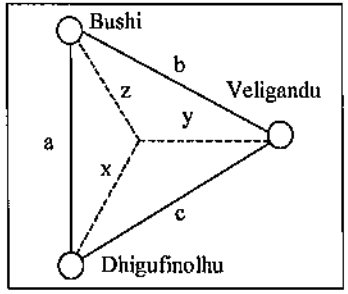


Figure 1: The Dhigufinolhu problem

There are two obvious solutions to this problem. The *abc* solution connects two of the islands at a time and the *xyz* solution provides a star-like connection. The *xyz* solution is thus cheaper to develop[3] since the accumulated length of the bridges is smaller. The *abc* solution will be cheaper over the lifetime of the bridges because the ways between the islands are shorter. This means that fewer employees are needed because they spend less time walking on the bridges.

Figure 2 shows the accumulated costs of the bridges. The figure shows that the quality of the design also depends on the *lifetime of the bridges*. If the bridges are used until T1, *xyz* is better, if they are used until *T2*, *abc* is better.

[3] We assume that the development costs for the bridges depend linear on the length of the bridges.

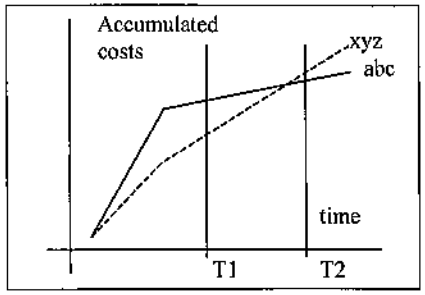


Figure 2: Costs of bridges

This example shows that:

Design ideals exist ...

1. We are able to find a design ideal if there are few variables (in our case the *costs* of the bridges over their *lifetime*) and
2. all variables are known.

These two conditions are nearly never met in real world problems. If we look at the actual solution of the problem, we see that neither *abc* nor *xyz* was chosen.

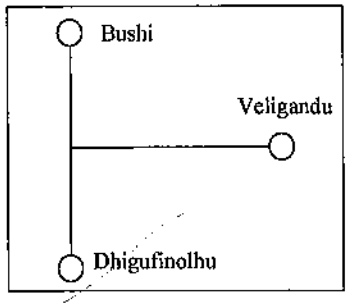


Figure 3: The reality

... only on the green field.

The development costs of this solution were higher than the optimal development costs and there are some long ways. Why choose such a solution? The answer is that the islands Dhigufinolhu and Veligandu are the tourist islands and Bushi is the island with the apartments for the hotel personnel as well as some infrastructure like fresh water tanks and generators. There is not much traffic between Dhigufinolhu and Veligandu (and it is *tourist traffic*). There is heavy traffic between Bushi and the other islands (especially Dhigufinolhu) mainly by hotel personnel. So we can see that the actual construction of the bridges is a good design trade-off. Some of the observations of the Dhigufinolhu problem can be transferred to the more abstract problem of software design and helps to understand the two main design support approaches of this thesis (described at the end of this section).

A good design has to fit well into a context. A context is a set of *given forces.* A force is an argument that influences the design. In software design, the most significant forces are related to costs. The lower the overall cost of a design the better the fit, the better the design.

An ideal design would fit ideally into its context. Fitting ideally into a context means that a design *resolves* the forces so that the sum of the related costs is minimized. Resolving a force means to provide a design that is not in conflict with the force. An ideal design minimizes the sum of all related costs. Often, the forces in a context are contradictory; e.g. a context may require minimizing the costs for analysis *and* the costs for maintenance at the same time[4]. An ideal design in such a context *balances* the forces so that the related costs are low. Balancing forces means to provide a design that minimizes the conflicts between the forces.

To *find an ideal design* we must entirely understand the *context* and the *relationships between context and design*. In software design the context *and* the relationships between design and context are poorly understood. The main problem is that a designer has to predict all contexts that may occur during the lifetime of the design. The forces in these contexts include the views of different (changing) users and different (changing) development and run-time environments.

We conclude that *an ideal design might exist, but it is not possible to recognize it because future forces cannot be predicted exactly.* This thesis suggests two main approaches helping designers to *understand the context* and *balance the forces*:

The first approach is to *make the design context more transparent.* A transparent context is one that is clear and well understood. The transparency of a context can be improved using measures and heuristics. We propose using measures and heuristics to represent important design arguments in a precise manner. A transparent context is also the key pre-requisite to better understand the relationships between design and context.

The second approach is to *provide tools that make predictions and approximations of future forces based on historical experiences.* Rules like the heuristics proposed here are intended to express forces which are present in almost every context. Therefore, they allow predicting a subset of the actual context based on experience. They are no forces per se! They help designers to make these forces (and their relationships to the design) more transparent.

[4] Graham, I. (1995). Migrating to Object Technology. Workingham UK, Addison Wesley. (p. 404) notes that "Process metrics can be related to each other, e.g. low analysis costs imply high maintenance costs".

1 Introduction

What is good software?

On a high level, there is an agreement that good software must *meet all user requirements*, have *low development costs* and be *easy to maintain*. It is also agreed that designing such software is a very hard task.

Software patterns as problem solvers.

A relatively new kind of support for this task is the published software *design knowledge* that is based on design experience. The best known area is the software pattern literature (Beck, Johnson 1994; Gamma, Helm et al. 1994; Coplien, Schmidt 1995; Buschmann, et. al. 1996; Vlissides, et al. 1996) and especially (Fowler 1993; Hay 1995). These software patterns are certainly the most important achievement for software designers in the last years but valuable design advice can also be found in methodology descriptions and development handbooks of organizations. Patterns enable communication on a higher level of abstraction and provide reusable (high quality) design fragments. What is more important, they show that it is possible to transfer design knowledge very efficiently.

Often, software patterns act as elements of a good initial design. However, it is still a hard task to find problematic locations in an existing design *that needs to be refined* (maybe through a pattern!).

Refinement is needed.

In this thesis, we are particularly interested in *refinement techniques* for *objectbases*. An objectbase is a collection of objects used by different clients. It is agreed that the schema of an objectbase is harder to design than that of a traditional database. The problem domains are more complex, the schema definition language is more complex (the designer has more choices), the shared objectbase has to fulfill most of the requirements of the traditional shared databases and additional properties like flexibility and reusability are desired!

Currently, refinement techniques are not included in design methodologies for objectbases. However, these techniques have always been included in relational database design methodologies to improve the database schema and achieve required database properties. Especially, algorithmic-based database design methodologies (Teorey, Yang et al. 1986) were designed to start with a "bad" relational database schema that could be improved later.

Therefore, it suggests itself to borrow from conceptual (Batini, Ceri et al. 1992) and algorithmic (Bernstein 1976) database design methodologies and re-vitalize them in an object-oriented context like, e.g. (Navathe 1989; Andonoff, Salaberry et al. 1992; Spaccapietra, Parent 1992). Unfortunately, the algorithmic and conceptual design methodologies have never been a complete solution – even for relational database schemas (which are nearly always optimized and denormalized to meet the performance requirements of the database!). An equivalent to relational database design methodologies for objectbases is needed that also considers operations, inheritance and other object-oriented features. The design of such a methodology is a challenging task.

The objectbase schema is hard to design since it has to fulfill many requirements..

1.1 Challenges of an objectbase design methodology

A closer look at existing object-oriented design methodologies shows that most of them define an *object-modeling language* or object specification language, a *graphical notation* corresponding to the language and a high-level *design process*. The language and the notation describe what a syntactical correct schema is (and how it is represented), but correctness does not imply "quality". The high-level design process is a specification of a sequence of design activities. It describes *which* activities are to be executed *when*. Therefore, it influences the used software process quality (at best!) but not the quality of produced schemas. This means that important ingredients (for designing a good schema) are missing in object-oriented design methodologies and have to be provided for objectbase design. However, a pragmatic approach cannot include a new and complete design methodology. The commercial methodologies are already frequently used, tools are available and with UML (OMG 1997) the modeling language is already consolidated – extensions are needed.

Challenge 1: What is a "good schema design"? Can current methodologies be *extended* to support the design of good schemas?

The success of the pattern community shows that canned design knowledge can efficiently be *transferred* to designers. On the software methodologist side there is also an agreement that a software methodology should include concrete knowledge and advice for good design (Henderson-Sellers 1995; Rumbaugh 1995). Unfortunately, the current approaches (Coad, Yourdon 1990; Henderson-Sellers, Edwards 1990; Arnold, Bodoff et al. 1991; Rumbaugh, et. al. 1991; Champeaux, Lea et al. 1992; Coad, Yourdon 1992; Booch 1993; Graham 1993) do not explicitly include design knowledge. Therefore, a methodology extension is needed that *provides* this knowledge.

Challenge 2: What is design knowledge? In which form do we provide it?

Obviously, it would be convenient to have a catalogue of design knowledge for all areas of software design. It was the starting vision of the pattern community to have such a software design handbook. Nowadays, we are far away from such a handbook. The deeper we understand the problem, the better we see how far we are away from the solution. Therefore, this thesis is restricted to *objectbase design*. The objectbases have the potential to play a similar role as traditional integrated database systems. Experience shows that an objectbase that acts as a "better programming language with persistence" for single user applications is relatively easy to design. However, designing an objectbase that fulfills all the requirements of a shared central integrated system is a very hard task. Unfortunately, a methodology that plays a similar role for objectbases as the traditional database design methodologies do for relational databases does not yet exist.

Challenge 3: What is special in the design of objectbases? Which *kind* of design knowledge is needed?

After having defined a methodology extension for objectbases, we have to ask ourselves how to provide it to the developers. Adaptation of such new technologies is a slow process. Incorporating the technologies in tools, which are already broadly used, can accelerate it. Finally, current pattern and heuristic approaches show that the design knowledge available today (in book and paper form) is already overwhelming and hard to manage "manually". Therefore, an approach that claims to efficiently support designers has to provide some kind of design tool.

Challenge 4: How can we support the methodology extension with tools?

Summarizing, the challenge is to provide a methodology extension for objectbase design methodologies. This extension has to make design knowledge tangible for designers. The design knowledge must include advice on how good objectbases can be designed. It must be structured in an orderly manner so that a design support tool can support the application of this knowledge. Such a tool has to be provided. Finally, the approach has to be validated in a realistic context.

1.2 Towards a design methodology extension

In this thesis, the focus is on a methodology extension for the object-oriented design of objectbases based on design knowledge. The objective of MeTHOOD is to *"distill"* this knowledge and provide it in a systematic design process that supports continuous review and improvement of a conceptual objectbase schema. MeTHOOD provides an *extension* to existing methodologies based on a catalogue of design knowledge. The catalogue combines and formalizes existing and new *design heuristics*, *transformation rules* and *measures* and adds them to the available design methodologies in an efficient manner. That catalogue is the main focus of this thesis.

A *design heuristic* (Riel 1995; Riel 1996), e.g. "Keep related data and behavior in one place", is a guideline or rule of thumb representing design experience and best practice. It is an advice for making design decisions. A design heuristic is used to describe a family of *potential design flaws* and helps designers to detect and avoid them. A potential design flaw is a fragment in a schema, which violates design rules, e.g. "related data and behavior are separated". Explicit descriptions of heuristics are needed. In most of the literature, design rules are implicitly described and hardly recognizable as useful heuristics ((Riel 1996) is the only attempt to make them more explicit). Therefore, one of the most important objectives of MeTHOOD is to explicitly provide important heuristics. In fact, the catalogue contains twenty-four explicit heuristics so far. These heuristics include different kinds of decision support for the correct use of single modeling concepts (e.g. selection of a complex attribute vs. a new class to model an address) and for the detection of potential design flaws.

If a potential design flaw is detected, often a *transformation rule* can be used to create a proposal for an alternative design. A transformation rule is an advice on how to eliminate a design flaw. For example, if a heuristic detects a location in the design where "related data and behavior are separated", a corresponding transformation rule suggests how to bring this data and behavior together. Often, there are different possible ways to eliminate a detected design flaw. Therefore, every design heuristic can have several different transformation rules associated with it.

Design heuristics and transformation rules allow to base design decisions on a clear rationale. However, there are two problems in this approach:

1. Different heuristics can conflict with each other. The first heuristic may require to "Keep all attributes of a class in its private part" whereas another one demands: "Values which are displayed in a user interface should be represented as public attributes of a class".
2. A heuristic can suggest different transformation rules to resolve a single design flaw.

How do we decide between these conflicting design advices? MeTHOOD provides object-oriented *measures* (sometimes also called metrics) (Abreu, Carapuca 1994; Chidamber, Kemerer 1994; Lorenz, Kidd 1994; Henderson-Sellers 1996) to give support for this kind of decision. A measure is a mapping between an entity and a number to characterize a feature (Fenton 1991) of the entity (e.g. the number of its dependencies on other entities). MeTHOOD measures are product measures. Designers use measures to *compare the effects of different heuristics and transformation rules on the design*. The measured values are used to recognize improvements resulting from changes in a design. They are monitored continuously during the com-

plete design process and provide designers with instant feedback on design decisions.

The integration of *measures, heuristics* and *transformation rules* is described in the MeTHOOD *integration schema*. That schema is a description of the relations between these concepts. The integration schema is the basis of the *MeTHOOD catalogue*. The catalogue contains

- for each measure, a list of heuristics detecting locations where the measure can be improved,
- for each heuristic, a list of transformation rules describing ways of how a violation of the heuristic can be removed and
- for each measure, a list of transformation rules resulting in improvements of the measure.

This knowledge integration framework for design support can be used in all areas of software design because the problem domain of the knowledge provided has no influence on the framework. The knowledge contained in the catalogue has been specifically developed and selected in order to support the design of objectbases (e.g. an order database in a stock trading system) in companies such as banks or insurers. In this type of environment, different project teams control multiple systems, and the main functionality is data retrieval, modification and presentation.

1.3 Restrictions and Assumptions

MeTHOOD assumes that a design methodology supporting class schemas is already in use. Without such a methodology, most of the material presented here cannot be used. Besides this, we assume that the design results are captured in schemas with a uniform *level of detail*.

MeTHOOD supports different levels of detail. A *highly detailed* class diagram contains

1. an attribute type for every attribute
2. a return type for every operation
3. a parameterlist for every operation with types for every parameter
4. a protection definition for every attribute and operation (private, protected, public)
5. cardinalities for every relationship
6. a collaboration specification for every operation, defining which operations of which classes are used.

A *medium detailed* schema omits the types of attributes, operations and parameters in 1)-3). MeTHOOD then assumes that

- none of the attributes and operations has a type dependency on another class in the schema
- attribute types and types of parameters are the same
- all return types of operations are the same.

A *low detailed* schema is a medium detailed schema without 4)-6). In addition to the assumptions above, MeTHOOD then assumes that

- all properties of a class are public
- all relationships are *n* to *n* relationships and
- there is no cooperation between classes in the schema.

The more detailed the schemas are the more precise results will be obtained from the measurements, and the more potential design flaws are detectable by heuristics. MeTHOOD has to assume that all elements of the same schema have the *same* level of detail. If two classes of the schema have different levels of detail (e.g. one has specified attribute types and the other has not) the measures will not be comparable and thus useless.

1.4 Outline

The remainder of this thesis is structured in four parts:

Part I introduces *the context* of MeTHOOD. It contains definitions of used terms and describes the used modeling language as a metamodel with a corresponding graphical notation. Furthermore, it describes the general approaches in the area of object-oriented methodologies, database design and objectbase design and concludes with some gaps that motivated the development of MeTHOOD.

Part II contains the *conceptual frame of MeTHOOD*. It describes a general design process and the essential requirements for a methodical approach to objectbase design. It introduces our view of measures, heuristics and transformation rules. These concepts are the base for the MeTHOOD integration which describes how implicit relationship between measures, heuristics and transformation rules can be made explicit and exploited for design. Furthermore, Part II includes a description of the MeTHOOD process, which consists of the design activities made possible by the MeTHOOD integration.

Part III is the core of this thesis. It contains *a catalogue of measures and heuristics* as practical advice for designers. The measure catalogue is based on a size and a dependency model. It contains a theoretical validation of the proposed measures. The core elements of the heuristics in the heuristic catalogue are the rationale

(describing why the heuristic should not be violated) and the checking rule (describing precisely how violations can be found). Part III concludes with a description how the catalogues help in objectbase design and how they are supported by a new design tool called MeX.

Part IV contains a validation of MeTHOOD. It contains a *differentiation to related work*. It allows validating MeTHOOD against similar approaches and shows the main similarities and differences to these approaches. Finally, it contains a *practical validation of MeTHOOD*. This validation describes the main results of experiments that apply MeTHOOD to design specifications of existing object-oriented systems. The number of classes of these systems varies from 200 to 1500 and they are developed with different object-oriented programming languages. The validation includes a first empirical validation of the proposed measures and shows the benefits and the limitations of this heuristic approach.

1.5 Summary of contributions

This section contains an overview of the achieved contributions in this thesis.

The first contribution is the *MeTHOOD integration*. It defines the relationships between the concepts *measure*, *heuristic* and *transformation rule*. This integration

- *eliminates some serious weaknesses* of these concepts (e.g. the passiveness of measures, the decision problem of heuristics) and
- allows to *represent* this design knowledge in an interrelated and effective manner, making it useful for designers and automatic design tools.

The second contribution is the definition of the *MeTHOOD process*. It shows how designers can *iteratively use* the MeTHOOD integration for systematic design improvements. This simple yet powerful process supports *measuring the existing situation, finding potential design flaws, proposing transformations for them, executing selected transformations* and *measuring the result*.

The third contribution is the *MeTHOOD catalogue* providing a set of important measures, heuristics and transformation rules for object-oriented design of objectbases. This is the main contribution of the thesis since the complexity of these objectbases increases, they play a more and more important role in many large systems, and design support is hard to find.

The fourth contribution is the *architecture and implementation of a design support system* which supports semi-automatic design improvement.

The fifth contribution is the design of MeTHOOD as *methodology extension*. MeTHOOD can be integrated into existing methodologies because it has the following features:

- it is based on its own metamodel which can be mapped to available notations (e.g. UML)
- it contains a systematic design process which can be integrated into available processes.

In summary, the *main contribution of this thesis* is a knowledge representation technique and a design knowledge base for objectbase design. It can be used to extend *existing* methodologies and enables

- a systematic design quality improvement process and
- tool support based on design knowledge.

It allows designers to

- specify explicitly desired quality parameters and better understand their interrelationships
- improve objectbase designs according to well-defined guidelines and patterns and
- use a design support tool which proposes design alternatives, transforms and documents a design.

Part I.
The Context

This part contains the basic definitions used in this thesis. Furthermore, it provides the context in which MeTHOOD has been developed: object-oriented methodologies, data modeling and objectbase design. The description of these areas concludes with specific gaps motivating the development of MeTHOOD.

2 Definitions and basic concepts

2.1 Modeling Terminology

The terms *schema*, *model* and *metamodel* are not uniformly used in current literature. Often, the same term, e.g. model, is used to describe different things like a modeling language (e.g., the relational model) or the model of a business domain[5].

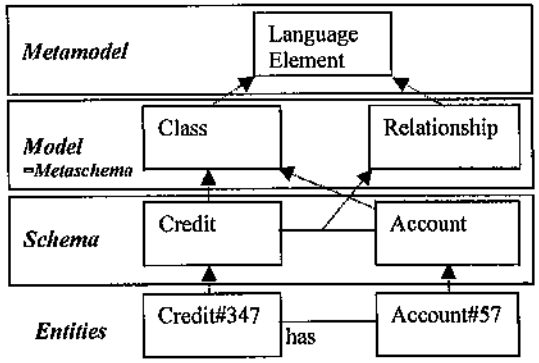


Figure 4: Metamodel, Schema, Model

Consider Figure 4: In a business domain, e.g. private banking, the *Account* with the account-number *57* and the *Credit* with the credit-number *347* are *entities*. The entity set of all accounts of *Town Bank* and its relationship to the set of all credits of Town Bank is *described* in the *schema* of Town Bank.

[5] In the database community, the term *schema* has the same meaning as the term *object model* in the object-oriented methodology community.

A schema is defined using a *model* (e.g. the Entity Relationhip (ER) model (Chen 1976)). The model includes various model elements used to describe entities in the schema (e.g. the ER model contains the model elements *Entityset* and *Relationship).*

If it is necessary to describe different *models* in a uniform manner, a further description language is needed. This description language is described in a so-called *metamodel*. The metamodel contains classes like *LanguageElement* and *LanguageElementRelation*.

Thus, the used modeling terminology in this thesis consists of the following four description levels:

Table 1: Modeling Terms

UML Term	Our Term	Description Level	Examples
meta-metamodel	Metamodel	Description of *model* or modeling language concepts	The repository schema of a case tool
metamodel	Model	*Description* of *schema* or modeling language	the ER model, the UML metamodel
model	Schema	*Description* of entity sets or classes of *objects*	Accounting Schema, Portfolio Schema
user objects	Objects	Description of real world entities	Account#47, Credit#347

The used separation of description levels is similar to the Four-layer metamodel architecture of UML (OMG 1997b) (Booch, Rumbaugh et al. 1996).

2.2 Conceptual design schema and design fragment

A *conceptual design schema* is a high level schema that is independent of languages of the implementation environment (e.g. a Data Definition Language (DDL)). It consists of a set of classes and their relationships. Other terms with a similar meaning include e.g. *logical design*, *architecture*, *object model* and *conceptual model.*

A *design fragment* is a subset of a conceptual design schema. It thus consists of a subset of classes with their structural (e.g. inheritance and associations) and dynamic (e.g. collaborators and messages) relationships. However, it does not include the relationships with classes outside. The *empty design fragment* ε is a special case. It contains no classes at all.

2.3 Model and graphical notation

MeTHOOD is a *methodology extension*. Thus, we assume that a methodology with a model is already in use. In this section, we define the set of language elements that are used in the MeTHOOD model. We need these language elements because we want to make our extension methodology-independent. MeTHOOD users can map the MeTHOOD model to the model of available methodologies.

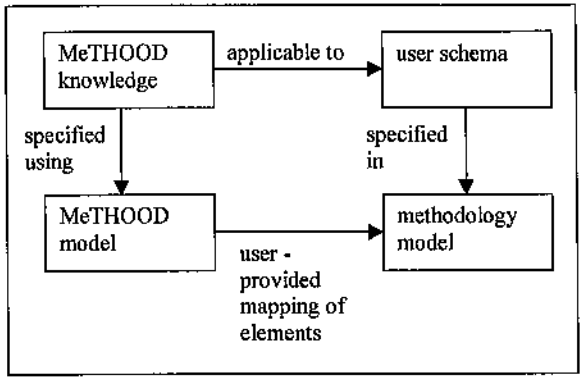


Figure 5: Metamodel mappings

Figure 5 illustrates this mapping. The knowledge provided by MeTHOOD is specified using the MeTHOOD model. This knowledge is applicable to the domain-specific schema provided by users. This schema is specified in the model the used methodology. To apply the MeTHOOD knowledge, the user has to provide a mapping between the MeTHOOD model and the methodology model. The MeTHOOD model is similar to the model of UML (OMG 1997b) and OML (Firesmith, Henderson-Sellers et al. 1998) and the mapping is straightforward.

2.3.1 Metaobjects

The language elements in the MeTHOOD model are presented as *objects of a metamodel*. These objects are called *metaobjects*. The MeTHOOD metamodel consists of the two classes *metatype* and *metarelationship*.

2.3.2 Metatypes

Table 2 gives an overview of the MeTHOOD metatypes. It contains the objects of class *metatype*:

Table 2: Metatypes

Metatypes
Schema*
Class*
Attribute*
Operation*
Message*
Type
Simple Type
Tupletype
Collectiontype
Classtype

Schema

A schema is a named logical grouping of classes and other schemas (subschemas). It consists of a name, *a set of classes, their relationships,* and a set of other schemas called *subschemas*. The same class and the same subschema can be present in different schemas. The name of a subschema or a class within a given schema is unique. The relationship between schemas, their subschemas and their classes is described in section 2.3.3.

Class

A class is an abstract description of a set of *similar entities*. It consists of a name and a set of properties. A property is either an *attribute* or an *operation*. A class can have different relationships to itself and to other classes. These relationships are described in section 2.3.3.

Attribute

An attribute is a part of a class. It describes a property that every object of this class has. Such an object has a value for every attribute. Every attribute has a name (e.g. *AccountNumber*) and an attribute type (e.g. *integer*). The name of an attribute is unique within its class. An attribute type can be a simple type like *integer*, *string* or *character*, or a classtype (see section 2.3.3.).

Operation

An operation is a part of a class. It describes a service provided by the objects of its class. An operation has a name, a *list of parameters*, a *returntype* and a *list of messages*. The name of an operation is not unique within its class. The signature (i.e. combination of operationname, parameterlist and returntype) must be unique. The parameterlist of an operation is a description of the information, which can be provided by clients of the service. A parameter in the parameterlist has a name and a type. Combinations of names and types in a parameterlist are not unique. A parameter in a parameterlist can only be identified by its position in the list. The return type can be a simple type or a classtype. The set of messages describes what different operations can be used by a given operation. The returntype of an operation is also called the type of an operation.

* Metaobject with explicit graphical notation

Property[6]

A property is either an *attribute* or an *operation*. We distinguish between *class and object properties*. Class properties are defined for the class, object properties are defined for single objects. Furthermore, we allow defining the *visibility* of properties. The visibility of a property is a description that regulates from where the property can be accessed. We distinguish between private, protected and public properties. A private property can only be accessed from within its class. A protected property can be accessed from within its class and from subclasses (see section 2.3.3.) of its class. Access to public property of a class is not restricted.

Interface/public interface of a class

The interface of a class is the *set of all properties* of the class. The public interface is the set of all public properties of the class.

Message

A message is a description of a *property call*. A property call is either an operation call or an attribute call. An operation call executes an operation. An attribute call retrieves the value of the attribute. Every message has a *supplier name*, a *message name* and an optional *message parameterlist*. We distinguish between *class messages* and *object messages*.

The supplier name of a *class message* must name an existing class (the supplier of the message). The message name must be the name of a *class property* of this class. The message parameterlist, if there is any, must match the parameterlist of a class operation with this name.

The supplier name of an *object message* must be either the name of a property of its class or the name of a parameter of the operation. The type of the property (attribute type or operation return type) respectively parameter must be a classtype.

The message parameterlist, if there is any, must match the parameterlist of an object operation with this property name. This means, the supplier of the

[6] The cursive text denotes terms that are introduced for convenience. The terms are not included in the metamodel.

message has an operation with a parameterlist, which *corresponds* to the message parameterlist. "Corresponds" means that the types of the operation parameterlist are compatible with the types of the message parameterlist. We use the same type compatibility rules as JAVA (Cornell, Horstmann 1996; Flanagan 1996; Gosling, Joy et al. 1996).

Example 1. Schema, Class, Attribute, Operation, Message

```
schema FinancialTransactionAccounts{                    //schema
 class FinancialTransaction{                            //class
  Account myAccount;                                    //attribute
  void transfer_money(int sum, Account receiver){       //operation
   myAccount.withdraw(sum);                             //message
   receiver.deposit(sum);                               //message
  }
 }
 class Account{                                         //class
  int deposit(int sum){                                 //operation
  ...
  }
  int  withdraw(int sum){                               //operation
  ...
  }
 }
}
```

Type

A type (sometimes called domain) is a named description of a set of possible values. We distinguish between the type of an attribute, the type of a returned result of an operation and the type of a parameter. A type can be a simple type, like *integer*, a tupletype, a collectiontype or a classtype. The terms type and class are not uniformly used in current literature.

Simple Type

A simple type describes is a type which is explicitly available in a construction environment. It is not further specified. Examples of simple types are *integer* or *string*.

Tupletype

A tupletype is a grouping of other types. A tupletype has a name and a set of type attributes. A type attribute has a name and a type. A name in the set of type attributes is unique. The name of the type attribute is used to identify a type in a tupletype. The type of a type attribute can be any other type.

Collectiontype

A collectiontype is a description of a set or a list of attributes or values. It has a name and a set type. The set type can be any other type.

Classtype

A classtype describes a domain consisting of a set of objects. The name of a classtype must be the name of an existing class. The domain of a classtype is the set of all potential objects of this class.

Example 2. Simple Type, Tupletype, Collectiontype, Classtype

```
class Customer{
  Address myAddress[String street, int number];   //tupletype
  Set(Account) myAccounts;                        //collectiontype
  void evaluate(Date theDate){                    //simple  type,  classtype
  }
}
```

2.3.3 Metarelationships

A metarelationship in MeTHOOD is a specification of a binary relationship between objects of class *metatype* (see Table 2). Table 3 contains an overview of the metarelationships in MeTHOOD.

Table 3: Metarelationships

Metarelationship	Source Metatype	Target Metatype
HasSchema	Schema	Schema
HasElement	Schema	Class
Has*	Class	Class
HasParts*	Class	Class
HasBaseClass*	Class	Class
HasAttribute	Class	Attribute
HasOperation	Class	Operation
HasType	Attribute	Type
	Attribute	Tupletype
	Attribute	Collectiontype
	Attribute	Class
HasParameter	Operation	Attribute
HasReturnType	Operation	Type
	Operation	Tupletype
	Operation	Collectiontype
	Operation	Class
SendsMessage*	Operation	Message
HasClass	Message	Class
UsesAttribute	Message	Attribute
UsesOperation	Message	Operation
HasTupleAttribute	Tupletype	Attribute
HasBaseType	Collectiontype	Type

Each of the metarelationships has a different meaning, described below:

HasSchema
specifies the relationship between a schema and a subschema. The schema and the subschema must exist. A subschema can be present in different schemas.

HasElement
specifies the relationship between a schema and a class. The schema and the class must exist. A class can be present in different schemas.

Has
specifies the relationship between a class and a referenced class. The class and the referenced class must exist. A class can be referenced by different other classes. The relationship can be uni-directional and bi-directional.

HasParts
specifies the relationship between a class and a contained class. The class and all the contained classes must exist. Each class can be contained by only one class.

HasBaseClass
specifies the relationship between a class and a base class. Both classes must exist. Every given class can have different base classes. The public and protected properties of the base class are also available in the given class.

HasAttribute
specifies the relationship between a class and an attribute. The class and the attribute must exist.

HasOperation
specifies the relationship between a class and an operation. The class and the operation must exist.

HasType
specifies the relationship between an attribute and a type. The attribute and the type must exist.

HasParameter
specifies the relationship between an operation and a parameter. The operation and the parameter must exist.

HasReturnType
specifies the relationship between an operation and a return type. The operation and the type must exist.

SendsMessage
specifies the relationship between an operation and a message. The operation and the message must exist.

HasClass
specifies the relationship between a message and its supplier. If the supplier is a class message, the supplier is the class. If the supplier is an object, the supplier is the class of the object. The message and the supplier must exist.

UsesAttribute
specifies the relationship between a message and the attribute it uses. The message and the attribute must exist.

UsesOperation
specifies the relationship between a message and its operation. The message and the operation must exist.

HasTupleAttribute
specifies the relationship between a tupletype and an attribute. The tupletype and the attribute must exist.

HasBaseType
specifies the relationship between a collectiontype and its base type. The collectiontype and the base type must exist.

Example 3. Metarelationships with their objects[7] for Example 2

HasTupleAttribute:	*(myAddress, street), (myAddress, number)*
HasType:	*(street, String), (number,int), (theDate, Date)*
HasType:	*(myAccounts, Set)*
Has Type:	*(theDate, Date)*
HasBaseType:	*(myAccounts, Account)*
HasOperation:	*(Customer, evaluate)*
HasParameters:	*(evaluate, theDate)*
HasReturnType:	*(evaluate, void)*
HasType:	*(myAddress, Address)*

2.4 Graphical Notation

MeTHOOD uses a single diagram notation.

For most models in the literature (e.g. (Booch 1993; Graham 1993; OMG 1997b)), a graphical notation is provided. Most of these notations have multiple diagrams for design. The MeTHOOD approach is different to the approaches above, but similar to (Coad, Yourdon 1990; Coad, Yourdon 1992). We propose a *single* extended class diagram as design specification. In this

[7] For better readability, the example does not contain fully qualified names.

section, we illustrate the notation used in this document and describe why we prefer a single diagram technique.

2.4.1 Class diagram

MeTHOOD uses a simplified UML (OMG 1997b) class diagram with extended support for cardinality and messages. MeTHOOD provides:

- support for interval cardinality (e.g. 4..5) and
- support for messages on a lower level of abstraction in the class diagram (see below).

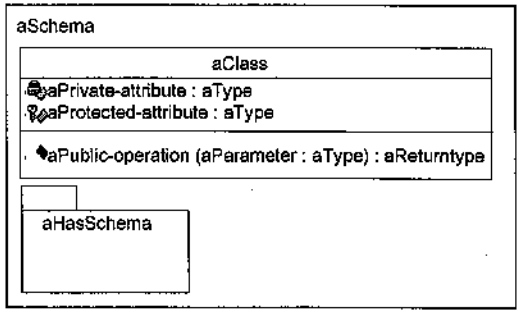


Figure 6: Schema, hasSubschema, Class, Attribute, Operation

Figure 6 shows the graphical notation for the metatypes *schema*, *class*, *attribute*, operation, and the metarelationship *HasSchema*. The metarelationships *HasElements, HasAttributes, HasOperations, HasType, HasParameters* and *HasReturnType* are implicitly expressed in this figure. Furthermore, the visibility of the properties is illustrated using icons.

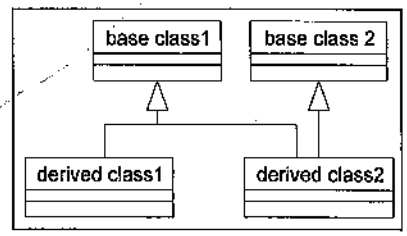


Figure 7: Notation: Baseclasses

Figure 7 shows the graphical notation for base classes and derived classes. The metarelationship *HasBaseClasses* is visible in this figure.

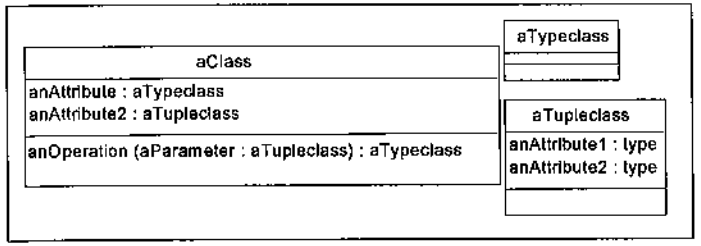


Figure 8: Notation: Typeclasses

Figure 8 shows the graphical notation for different types. The metatypes *Type, Tupleclass (attribute2),* and the metarelationships *hasTupletype (attribute2), HasClass (attribute) and HasTupleAttributes (Tupleclass)* are visible in this figure. The metaobject *HasTupleReturn* is equivalent to the visible *HasObjectReturn (operation).*

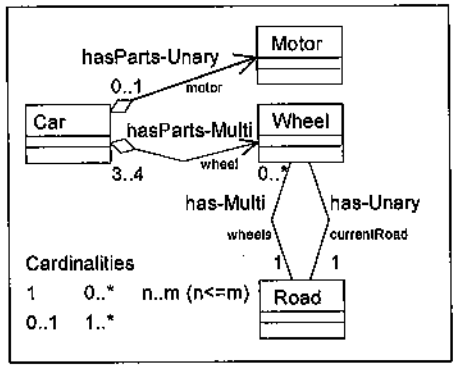


Figure 9: Notation: Associations

Figure 9 shows the graphical notation for the different kinds of associations. The metaobjects *Collectiontype (has-Multi)* and the metarelationships *Has, HasParts* (with different cardinalities) are illustrated in this figure.

Figure 10 shows the graphical notation for different kinds of messages. The metarelationships *SendsMessage (uses), UsesAttribute (value) and UsesOperation (add (Quantity))* are visible in this figure.

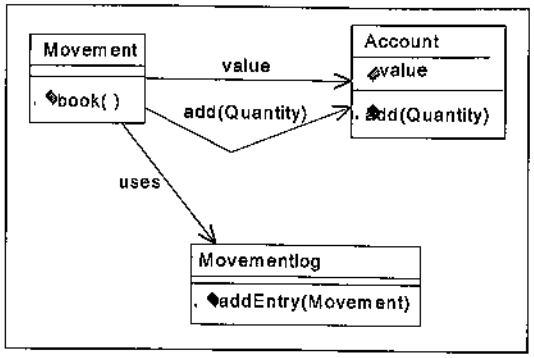


Figure 10: Notation: Messages

2.4.2 Other diagram types

In this document, we also use interaction (collaboration diagram, sequence diagram) (see Figure 11 and Figure 12). However, concepts from these models are not used in the MeTHOOD catalogues and in the MeTHOOD metamodel. The syntax and semantics of these diagram techniques correspond to syntax and semantics described in UML (OMG 1997b).

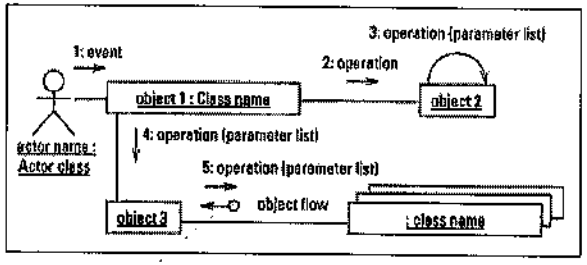


Figure 11: Collaboration diagram

Figure 11 shows a collaboration diagram. The collaboration is initiated by an actor that sends the event *event* to *object1* of the class *class-name*. This object executes further operations on other objects.

Figure 12 shows an UML sequence diagram (OMG 1997b), which provides the same information as Figure 11.

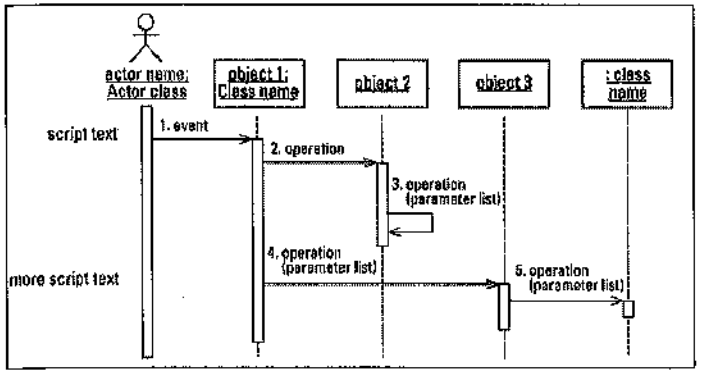


Figure 12: Sequence Diagram

2.4.3 Problems with multiple diagrams: complexity management

In the beginning, object-oriented methodologies included one single, or very few diagrams. The maturing of the field in the late 90ies resulted in a consolidation of the different kinds of diagrams, but unfortunately most of the diagrams *survived* (and are now present in UML (OMG 1997b)). Today, most methodologies propose using multiple diagrams that allow specifying different views on a problem. It is assumed that a single diagram would be too complex to provide the required information. We have observed that the usage of many different models is not the best technique to manage the complexity of a complex problem. On the contrary! The problem with these approaches is that the information provided in the different models is related. Often the same concepts play different roles in the different models.

Multiple diagrams introduce accidental complexity.

This means that the relation between the different roles that the concept plays has to be specified. Often the same concept has to be specified twice. Therefore, accidental complexity is introduced. Consider, for example, the class diagram and the object interaction diagram in UML (OMG 1997b).

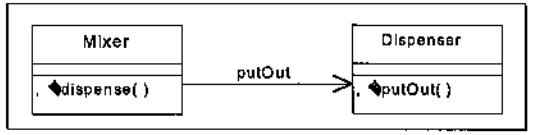


Figure 13: Single Class Model

Very often, we would like to state, an operation of one class can use another operation in another class (e.g. *dispense*() of the class Mixer can use *putOut*() of Dispenser). This is important since we need to know which changes in

the interface of a class propagate to other classes. This information cannot be provided in a single UML class diagram (OMG 1997b). Therefore, we have to specify a sequence diagram or a collaboration diagram to provide this information (Figure 14).

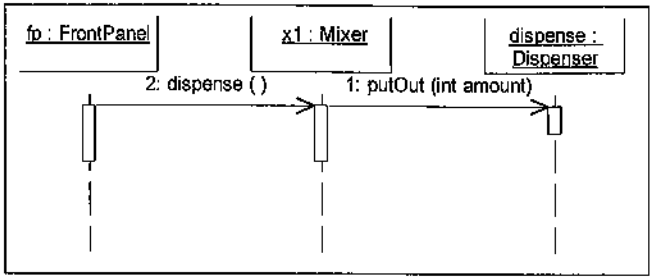


Figure 14: Sequence Diagram

The situation gets even worse if we try to find out which interactions can occur between the two classes. In this case, we have to look at *all* sequence diagrams and *all* collaboration diagrams.

MeTHOOD proposes a single diagram technique.

The basic specification technique of MeTHOOD is to use extended class schemas as a *single specification technique*! This idea is similar to the ideas of (Coad, Yourdon 1990; Coad, Yourdon 1992). MeTHOOD avoids using different specification techniques as a tool to manage the complexity of the problem. In a single diagram modeling approach, all available modeling constructs can be shown on a single diagram. The specific information that is visible at a specific point in time can be specified using filters. The message *putOut*() between Mixer and Dispenser would imply a dependency between the classes, and all potential messages between the classes could be shown.

3 Object-oriented Methodologies and Data Modeling Approaches

Objectbase design has its foundations in the areas of *object-oriented methodologies* and *data modeling approaches*. To understand how current methodology can be extended, it is necessary to understand the main requirements of objectbase design not covered in these foundations. Therefore, in this section we present the main concepts of *object-oriented methodologies* and *data modeling approaches* and conclude with a description of the gap to be filled.

3.1 High-level modeling techniques

Conceptual modeling and object modeling are high-level modeling techniques.

Object-oriented methodologies and *data modeling methodologies* include so-called high-level modeling techniques (also called conceptual modeling or object modeling) used to specify a schema on a high level of abstraction (and independent from an implementation language). It is a common pattern that the *modeling techniques* are developed *after* the implementation languages (or the data definition languages (DDLs)). This is true for object-oriented design methodologies developed in the early 80ies (*after* the development of programming languages like (Dahl, Nygaard 1966; Ingalls 1976; Wirth 1977; Ichbiah 1980)) and for the entity relationship model (Chen 1976) (and all of its extensions) which succeeded the relational model (Codd 1972).

The reason for this is that the development system (e.g. the programming language or the database system) is developed with a specific usage profile and a specific level of complexity in mind. If the system is successful, users start using it with different other usage profiles and on problems with a higher level of complexity. One of the problems is that the level of the implementation language is too low for the higher level of complexity. The solution for this problem is to introduce a *modeling language on a higher level* and map it on the implementation language.

Objectbase systems are currently in the phase where a higher-level modeling technique is needed. Raising the implementation language on a higher level is not useful, because one of the main arguments to use objectbases is that their implementation language is already on a sufficiently high level. Thus, the modeling language of current design methodologies is *not* on a higher level of abstraction than the implementation language – it merely provides a graphical notation. Completely different approaches are required to solve the subsequent problems. In order to investigate how these approaches could look like, we first take a closer look at modeling techniques from the database and the programming language perspective.

3.2 Object-oriented methodologies

A methodology is a collection of tools, activities and principles that help practitioners to execute tasks in a discipline. Many authors have observed that a characteristic of every emerging discipline is an explosion of methodologies and that for object-oriented analysis and design methodologies, this explosion has started in the early 80ies.

3.2.1 What should be included in a methodology?

The ingredients of a methodology ...

It is agreed in (Rumbaugh 1995) and (Henderson-Sellers 1995) that a methodology should include:

1. a modeling language,
2. a notation,
3. a process and
4. advice.

Besides these features, (Henderson-Sellers 1995) demands

5. a full description of deliverables,
6. quality advice,
7. guidelines for project management,
8. reuse organization and
9. organizational roles.

1. A *modeling language* is a set of concepts (abstract syntax) which can be used to define a specification. Examples of concepts are classes, attributes, objects, events etc.
2. A *notation* is a representation of the modeling language (concrete syntax). A notation can consist of a set of icons or a textual representation. A notation should be complete (all concepts of the modeling language should be expressed), easy to understand and supported by tools.
3. A *life cycle process* is a description of a set of activities, which should be performed in the methodology. It should be a clearly structured step-by-step guide for users of the methodology.

4. The required *advice* should help users applying the methodology. It should include a collection of hints, rules of thumb or guidelines capturing experience, best practice and best current theory.
5. The *deliverables* are the results of executing the activities in the process. They should be precisely described and related to each other, e.g. the analysis specification is an input for design.
6. The *quality advice* should include a set of metrics, advice how to achieve quality and test strategies.
7. The *guidelines for project management* should include information about scheduling the activities, project planning and coordination with other Projects.
8. The *reuse organization* should provide advice on project library management, reuse tools and organizational roles that facilitate reuse processes.
9. The *organizational roles* should describe the roles, which are necessary to execute object-oriented projects. Example roles include projectmanager, developer and project architect.

3.2.2 Evaluation of the required features

It is widely agreed that features 1.-4. should be included in every methodology. There are some arguments on the level of detail of the process (3.). It is inherently difficult to define a single sufficiently detailed process suitable to a lot of different types of projects. Some methodologists claim that a process framework or a process construction kit would be sufficient. The same is true for the description of deliverables (5.), project management (7.), reuse organization (8.) and organizational roles (9.). These items refine what other methodologists consider a process. Quality advice (6.) is important and actually required by many designers. It should be included in every methodology.

3.2.3 What is included in current methodologies?

Today, we look back on three generations of object-oriented analysis and design methodologies. The first generation of methodologies has been developed from the early 80ies on. The most important first generation methodologies are the Booch methodology (Booch 1991), the Rumbaugh methodology (Rumbaugh, et. al. 1991), responsibility driven design (Wirfs-Brock, Wilkerson et al. 1990), HOOD (HOOD 1990), the Buhr methodology (Buhr 1991), the Coad&Yourdon methodology (Coad, Yourdon 1990; Coad, Yourdon 1992), the Shlaer&Mellor methodology (Shlaer, Mellor et al. 1988), use case driven design (Jacobson, et. al. 1992) and the methodology of Colbert (Colbert 1989). Some of these early methodologies are tightly coupled to a particular programming language, e.g. HOOD (HOOD 1990) and (Buhr 1991). Nearly all of the methodologies include a modeling language and a graphical notation. Some of the methodologies include a high level process. It has frequently been noticed that the first generation methodologies are quite similar to each other. Some distinguishing concepts are *use cases* in (Jacobson, et. al. 1992), the *single diagram concept* (see section 2.4.3) in (Coad, Yourdon 1990; Coad, Yourdon 1992), the notion of

responsibility of an object in (Wirfs-Brock, Wilkerson et al. 1990) and the mapping of the *modeling concepts to relations* in (Rumbaugh, et. al. 1991).

The second generation of methodologies has been available from 1996 on. These methodologies are strongly influenced by the first generation methodologies. The most important of them are Fusion (Coleman, et. al. 1992) (Firesmith 1993), Catalysis (D'Souza, Wills 1997) and SOMA (Graham 1993; Graham 1995). The second-generation methodologies are (again!) quite similar to each other. Nearly all of them claim to incorporate the good ideas from the first-generation methodologies. (Graham 1995) can be distinguished from the other methodologies because the proposed SOMA methodology has a strong focus on object-oriented analysis (including model walk-throughs), and it provides rule sets on classes that allow specifying integrity constraints. SOMA and the other second-generation methodologies include an extensive treatment of the requirements 5., 7., 8., 9. The description and the level of detail of these features differ in all methodologies and are on a different level of abstraction. The most extensive treatment can be found in (Graham 1995).

The third generation consolidates first and second.

The third generation of methodologies are UML (OMG 1997b) and OPEN (Graham, Henderson-Sellers et al. 1997; Firesmith, Henderson-Sellers et al. 1998). These methodologies are a consolidation of all former methodologies. UML is a de facto industry standard because it is widely used and supported by most case tools. Modeling language and notation of both methodologies are quite similar. The treatment of the remaining issues, especially the process 3. and the features 5., 7., 8., 9., is much more extensive in (Graham, Henderson-Sellers et al. 1997; Firesmith, Henderson-Sellers et al. 1998). The process of UML is based on (Jacobson, et. al. 1992).

3.2.4 Design support in methodologies

There are three categories of object-oriented design methodologies with different types of design support:

1. Methodologies with implicit design advice and examples
2. Methodologies with explicit heuristics
3. Methodologies with metrics/measures

1. Most current methodologies contain example designs, e.g. (Booch 1993) and implicit design advice, e.g. "Avoid traversing associations that are not connected to the current class" (Rumbaugh, et. al. 1991). On the other hand, we can find explicit heuristics that give advice on how specific elements in the provided object model should be used.

2. Some recent methodologies like (D'Souza, Wills 1997) explicitly contain collections of heuristics. The heuristics we have found are often on a high level, like "Build a system that mirrors the real world; and keep it that way." These kinds of heuristics are intended to give strategic advice which is able

to move the whole design in a certain direction. Often, heuristics for the design *process* can also be found, e.g. in (Booch 1995).

3. Other methodologies contain a collection of metrics or measures, e.g. (Graham 1993). In most cases, these metric collections in methodologies are similar to (Chidamber, Kemerer 1994). All provided metrics help designers to assess the quality of their designs and provide hints on how designs can be improved, because each metric also implies advice how to improve its value (implied heuristic).

3.3 Data modeling

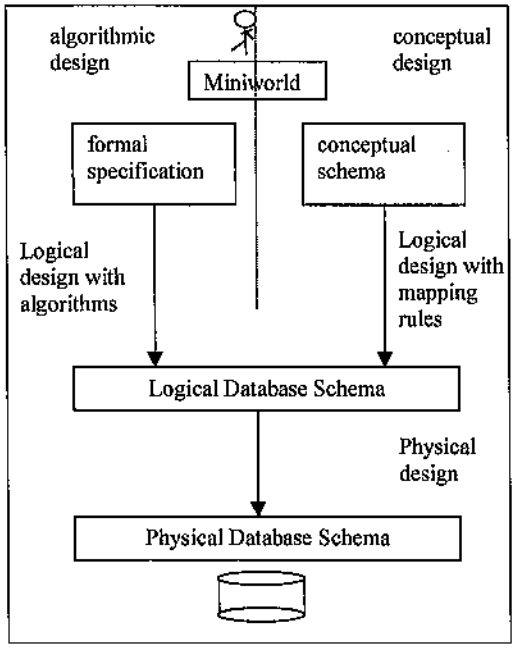


Figure 15: Data modeling approaches

Data modeling (or database design) is the process of determining the organization of a database, including its structure, content and the applications to be run (Batini, Ceri et al. 1992). Relational database design is a stable discipline which has become part of the general background of computer scientists (Batini, Ceri et al. 1992). It can be separated in the areas of *algorithmic database design* and *conceptual database design* (see Figure 15).

Algorithmic database design is an activity with the objective to produce a concrete database schema from a formal specification of a set of functional dependencies between attributes. Algorithmic database design activities start relatively late in the development process. A set of attributes is required as a prerequisite.

Conceptual database design is an activity in the earliest phases of the development process. The objective of conceptual database design is to produce a high-level, DBMS-independent schema, starting from requirement specifications that describe reality (Batini, Ceri et al. 1992).

Some approaches (Andonoff, Salaberry et al. 1992) also propose to *integrate* conceptual database design and algorithmic database design. Conceptual database design and algorithmic database design are the base for most *object-oriented database design* approaches. They have been developed parallel to the first and second-generation object-oriented methodologies. In this section, we look at the traditional database design approaches first and then we consider the object-oriented approaches.

3.3.1 Normal forms and algorithmic database design

The algorithmic approach to relational database design employs a formally defined model to detect and remove structures in a logical schema, which could lead to anomalies in the database. Anomalies in relational databases often occur when the database contains redundancies and when generic update operators are used (especially if the updates come from different client applications). To avoid certain types of redundancies, algorithmic database design is used. This approach is based on normal forms, functional dependencies between attributes (Housel, Waddle et al. 1979) and transformation algorithms (see (Maier 1983) or (Elmasri, Navathe 1989)).

Starting with a set of attributes or an universal relation (which can be considered as a very bad relational database schema) and a set of functional or multivalued dependencies, relational normalization algorithms generate relational schemata in 3rd or higher normal forms which are considered good relational schemata (Elmasri, Navathe 1989). The algorithms always have a part that checks if the intermediate schema is already in the desired normal form. If this is not the case, it finds the locations where schema transformations are necessary.

3.3.2 Conceptual database design

Conceptual database design is a design technique in which high level, DDL (data definition language)-independent design languages are used, e.g. the entity relationship (ER) model (Chen 1976). The main argument for this approach is that users are able to participate actively in the process that improves the schema quality and leads to better user acceptance (Batini, Ceri et al. 1992). Other strong arguments for conceptual design are that the choice of the specific DBMS can be postponed and changed (since the conceptual schema is DBMS-independent), the conceptual schema can "survive" requirement changes and different databases described in a conceptual schema (e.g. a hierarchical database and a relational database) can be compared.

Conceptual database design approaches have been used for a lot of systems, and soon after the publication of the ER model, two *types of extensions* have been proposed. The first type considers the *process* used for conceptual design. The second attempts to extend the ER model itself.

A design process describes different design transformations and shows when to execute which of them. A design transformation is a process that converts an initial design fragment into another fragment. (Batini, Ceri et al. 1992) distinguishes between *top down transformations* and *bottom up transformations*. Top down transformations transform a fragment to a lower level of abstraction. Bottom up transformations introduce new concepts or modify existing concepts wherever features are discovered that were not captured in the initial fragment. A conceptual design process is a mixture of these different types of transformations. (Batini, Ceri et al. 1992) proposes four different strategies to mix the transformations. In the top down strategies, a start concept is created, and only top down transformations are used. In the bottom up strategies, only bottom up transformations are used and merged. The inside-out strategy starts with a fixed set of most important concepts (bottom up) and then proceeds in a top down manner. In the mixed strategy both, top down and bottom up strategies are mixed, based on a partitioning of requirements.

Modifications of the ER model led to various extended ER models (EER models). The mandatory elements of an EER model are the basic features of the entity relationship model: entity, relationship and attribute. The extension of the ER model typically consists of different kinds of generalization and composition relationships between entities and composite attributes. Many of these extensions can be considered as essential for object-oriented database design, and the borderline between EER models and object-oriented ER (OOER) models is not clearly defined.

3.3.3 Object-oriented database design

The first kind of object-oriented database design methodologies provide OOER models. An OOER modeling approach seeks to integrate a set of object-oriented language elements into an ER model, e.g. (Lazimi 1989; Navathe 1989; Gorman, Cgoobineh 1991; Hörner, Heuer 1991; Ku 1991; Nachouki, Chastang 1991) and (Spaccapietra, Parent 1992). This kind of programming language/database modeling integration has a long tradition and can be subsumed under the term *conceptual modeling*[8]. The term conceptual modeling describes the modeling activities on a higher level (a conceptual level) of abstraction. It has been coined in the preface of (Brodie

[8] Nowadays, the terms *conceptual design* and *conceptual modelling* are used as synonyms.

1982) as an attempt to emphasize the common goals *of knowledge representation in artificial intelligence (AI), semantic data models in database design and abstraction in programming languages*. Whereas the influence of AI techniques on current object modeling approaches is limited to a few techniques, e.g. rulesets in SOMA (Graham 1995), the integration of modeling techniques used in database design and programming languages have been extensively examined.

In fact, most conceptual OOER models have been developed parallel to the first generation object-oriented (programming language) methodologies (1987-1993), and many of these methodologies have been proposed and used for object-oriented database design, e.g. (Martin, Odell 1992), (Rumbaugh, et. al. 1991), (Loomis, Shaw et al. 1987). Therefore, the second kind of object-oriented database design methodologies are *software design methodologies applied for database design.*

A third kind is represented by approaches like (Andonoff, Salaberry et al. 1992) (Davis, Delcambre 1989) (Kim 1991). These approaches propose *different types of formal algorithms* for object-oriented database design. (Davis, Delcambre 1989) proposes several algorithms that infer *isa*-relationships implicitly specified in a schema. (Andonoff, Salaberry et al. 1992) transfers functional dependencies, multivalued dependencies (Fagin 1977) and decomposition and synthesis algorithms (Bernstein 1976) (well-known in the context of algorithmic relational database design) to an object model. The use of these types of algorithms as strict rules can be argued, e.g. (Graham 1995) comments that these rules to achieve (an equivalent to) the third normal form (3NF) (Elmasri, Navathe 1989) do not belong to the object model and should only be considered as a heuristic guide.

3.4 Gaps in current modeling techniques

3.4.1 The methodology advice gap

Some features are covered but there are many gaps.

Based on section 3.2.1 and section 3.2.3, we can conclude that the features 1. (modeling language) and 2. (notation) are relatively well understood, tested, included in all methodologies and consolidated. The elements of 3. (process) and 5. (deliverables), 7. (project management), 8. (reuse organization), 9. (organizational roles) are not well consolidated, but we can find coverage of this topic in (Jacobson, et. al. 1992), and advanced coverage in (Graham, Henderson-Sellers et al. 1997; Firesmith, Henderson-Sellers et al. 1998). (Graham, Henderson-Sellers et al. 1997; Firesmith, Henderson-Sellers et al. 1998) is the only methodology which includes treatment of 6. (quality metrics). The authors propose an extensive set of metrics which are based on the task points described in (Graham 1995) and the well-known approach (Chidamber, Kemerer 1994). Collection of these metrics is possible with the SoMATIC tool set (see Appendix A: SoMATIC). In addition to that, (Hong, Goor et al. 1992) observes that most of the development methodologies focus on greenfield development, and reuse and reengineering are hardly considered. It claims that "... the issues of testability, traceability, reusability and conceptual integrity are only lightly addressed, ...". Furthermore, we see that there is a consensus that a good methodology should contain heuristics and concrete design advice (Rumbaugh 1995) (Henderson-Sellers 1995), but the requirement 4. is not yet explicitly covered in any methodology. Implicit advice can be found, e.g. in (D'Souza, Wills 1997), but this advice is not tangible, hard to realize and not supported by tools. From the methodology perspective, this requirement represents the largest gap which should be bridged!

3.4.2 The need for integrated design knowledge

Based on section 3.2.4, we see that it is difficult to use the provided design knowledge in current methodologies in an efficient way because different methodologies contain different kinds of advice buried in their documentation. The design advice is not supported by tools (which would be possible, if we consider the tool set provided for the advice described in (Lieberherr 1992)). Most (explicit and implicit) heuristics in current methodology literature are on a very high level and hard to check. Descriptions of *how* a given design can be improved using the heuristics are not given. The heuristics are not related to metrics; an evaluation of the influence of a heuristic to the design is not possible. From the perspective of object-oriented methodologies, an explicit description of integrated design advice and tool automation of this advice is urgently needed!

3.4.3 The need for higher level modeling languages

As we have seen in section 3.2.3, most object-oriented methodologies have a *strong focus on the quality of the modeling language*. Some of them seem to assume: "If the modeling language is right, the design will be right". We also observe that the abstraction level of current *modeling* languages is not higher than the abstraction level of the current object-oriented implementation languages (in fact, some implementation languages, e.g. Eiffel (Meyer 1988), seem to provide a higher abstraction level than some modeling languages). This is a major difference to structured methodologies for procedural languages and conceptual design methodologies for relational database systems.

From the modeler's point of view, the situation is not desirable. In our opinion, it will take some time until we can expect that the modeling language will be put up back to a level of abstraction that makes a difference. *This new level of abstraction will be based on experience*! Current pattern approaches are certainly the most promising steps in this direction, but it is still unclear how a pattern based modeling language would look like.

A second area of approaches are *domain modeling languages*. A domain modeling language is one that includes domain specific modeling constructs

(e.g. currency, customer, etc). Some basic domain standards (the base for such a modeling language) are currently under development, but a domain specific modeling language is not yet in sight. We propose an intermediate step in this direction (integrating design experiences into the process) which can be realized very quickly. The following chapters describe this approach.

3.4.4 The need for multiple design criteria for objectbase design

The more criteria included the better the design.

On the surface, traditional algorithmic database design methodologies cannot be criticized: they are designed for a specific context (to achieve a good relational schema). Large parts of the context have been described formally (the relational model), the context is nearly transparent (see Preface: The Dhigufinolhu design problem), the quality issues seem to be relatively well understood, and techniques to optimize the quality are available. Therefore, transferring this "well-being world" to objectbase design seems to be desirable.

We have observed that for real systems, relational database design is hard. After various iterations of physical design, database system configuration and manual query optimization, database schemata are "de-normalized" to achieve the desired database application performance. The reason for these problems is that the schema properties considered by normalization algorithms are very limited (relational algorithms consider only avoidance of data redundancies and potential update anomalies).

Much more properties need to be considered in order to design a good database schema, but the "algorithmic idea" has inspired the development of MeTHOOD: Every design algorithm in relational database design has a part which checks if the examined schema already fulfills a normal form. This part of the algorithm can be compared to the checking rule of a heuristic that states compliance to the desired normal form. The heuristic itself can be compared to a normalization rule. Another part of the algorithm then transforms the schema so that the identified locations comply with the normal form. This part can be compared to a transformation rule. A major difference is that we propose *multiple* desired quality features (and techniques to fulfil them). Finally, the transformation rules we propose are not used to generate a complete design fragment automatically. They are used to provide decision support for the designer.

3.4.5 The need for loose coupling between object-oriented database technology and methodology

The usage profile of relational databases is relatively clear. They are used as databases accessed by applications! Therefore, it makes sense to use methodologies specifically designed for this task (see sections 3.3.1 and 3.3.2). A usage profile of object-oriented database systems does simply not exist. OODBMS are used as a convenient persistence extension for object-oriented programming languages, and they are used as autonomous objectbases. They can be used as a data store (for simple and complex structured data) with all of the operations defined on the client side. They can be used as an objectbase (for simple and complex structured objects) with operations on the server side. All these usage profiles require completely different design approaches. The situation in current object-oriented database research reflects this: There is no agreement how an objectbase should be modeled. Some approaches propose to use an extended entity relationship approach, object-oriented methodologists claim that their approaches can be used for objectbases, too, and objectbase vendors even claim that modeling gets easier (because now it is possible to map the real world), and special modeling techniques are not needed – and all of them are more or less right! E.g. proposing a *software* design methodology for designing an object-oriented *database*, which seems to be naive from a database perspective, can be perfectly justified if we design a simple, one-platform, single-user chess program. Therefore, the notation of "object-oriented database design methodology" is of limited use only. The different usage profiles suggest that the design methodologies should not be coupled to the usage of a database system but to the usage profile! Therefore, we choose the notation "*objectbase design*", which means *object-oriented software design for shared integrated multi-user objectbases (which are used by different applications).*

4 Objectbase design

Objectbase design is *software design* with the focus on designing good objectbases. Before we describe software and objectbase design in detail, we illustrate the general idea of design, which is common for all design activities.

Design consists of a variety of activities for the *invention* and the *optimization* of a form. A form is a representation of a set of ideas. Invention is a creative process with the objective to create a form fitting into a given context. For software design, the *form* is a design specification and the *context* is a set of requirements. Optimization is a process with the objective to achieve a better fit between a form and a context. Most design activities can be categorized into these two categories.

The context is the starting point for the complex process of building the system. In this process, *design* is an activity for describing *how a system should work*. Examples of systems are a software system (software design) or a business system, such as a set of retail business processes (business design). Analysis is an activity that investigates *what* kind of functionality must be provided by the system. A good analysis results in a common understanding of the required functionality of the systems between users and developers of the system. Traditional approaches (Royce 1970) attempt to make a clear separation between the two. However, many drawbacks of these approaches have been observed. Therefore, many organizations execute analysis and design iteratively. The results of analysis are the input for design. Analysis and design can be considered as object-oriented, if object-oriented techniques and description means (based on an object model) are used. We will focus on software system development.

4.1 Object-oriented software design

Object-oriented software design is an activity in the software system development process. This activity is performed to transform requirements (e.g. a description of a business system, which is the result of business analysis and design) into a *software model* (a description of a software system, which is the result of software analysis and design) (Figure 16).

Business analysis is performed in order to describe what the business is doing, e.g. provide damage insurance. *Business design* then describes how these responsibilities are implemented, e.g. introduction of organizational units, business processes and supporting software systems. *Software analysis* bridges business design and software design describing

1. user requirements (what the system has to do),
2. system requirements (concerning e.g. performance and reusability) and
3. business abstractions (e.g. customer, damage claim, ...) within such a system.

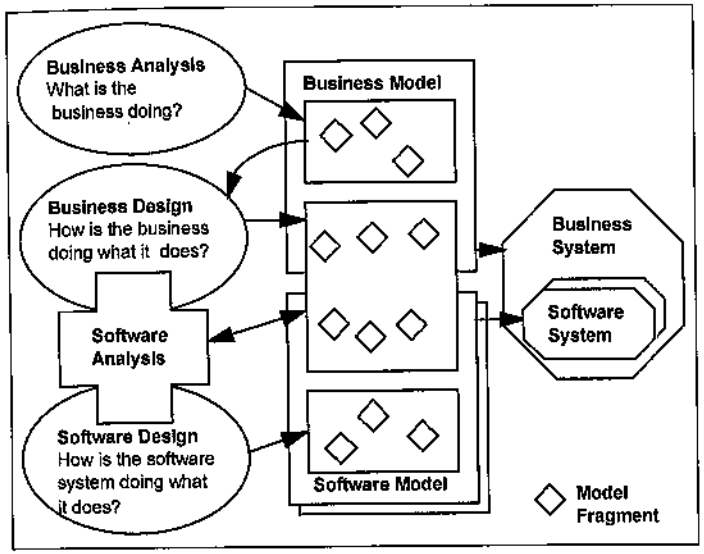


Figure 16: Process, models and systems

An example of a part of a business model is a description of a business process (e.g. "handle damage claim"), business resources (e.g. damage claim, customer) and organizational units (e.g. claims department). An example of a supporting software system is a damage claim system which receives damage claims from different sources, e.g. fax and email and distributes them to the employees. One user requirement is that "a damage claim is automatically routed to the right employee".

The main results of object-oriented software analysis are system and domain *requirement specifications* and specifications of *abstractions in the business domain* (e.g. damage claim and customer), *described using an object-oriented model*. These so-called *domain abstractions* (e.g. (Sutcliffe, Maiden 1994)) should be understandable for domain experts. The results of the

software analysis are the input for object-oriented software design (Figure 17).

The process of software design starts with the elicitation of *technical classes* (e.g. classes for database access), which are necessary for the implementation of the software system (realization analysis). Technical classes are classes which are understood in the software development domain. They typically cover version management, history, user management and user interfaces, and often represent existing systems.

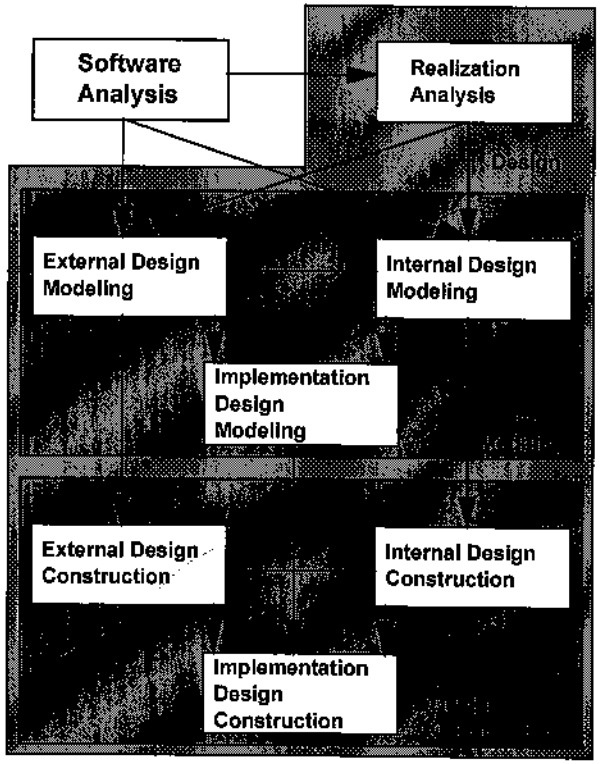


Figure 17: Object-oriented software design

Domain classes, technical classes and requirements are inputs for *design modeling* and *design construction*. Both activities are design activities. Whereas design modeling is invention *and* optimization, design construction mainly considers optimization for a given construction environment. A construction environment is a specific product, e.g. a programming language, a database management system or a set of libraries. The difference between design modeling and design construction is that the results of design *modeling* are independent of the selected construction environment, whereas the results of design construction are specified in the languages corresponding to this environment (C++, SQL, etc.). Design construction can also be considered as a mapping from a conceptual design model to the construction environment. Design construction corresponds to the transition from the logical view to the development view in (Kruchten 1995).

Design modeling and design construction are performed iteratively (see Figure 17), although design modeling should start before design construction. Both, design modeling and design construction, are subdivided into three areas:

1. *External design* considers a set of class interfaces. An interface represents properties of a class which are visible to other classes (attributes and operations). These interfaces can be interfaces of simple classes or classes which represent groups of other classes (class categories). A class within such a category can only receive messages from other classes in the category or from the class which represents the category. The result of the external design is a specification that includes class interfaces and the messages passing between them.
2. *Internal design* considers the definition and refinement of a single class based on its interface definition and the messages to be sent to other classes. Activities include definition of attributes and operations which do not belong to the interface and their relation to the interface.
3. *Implementation design* considers the full *definition of operations*, the given *operation declarations from external and internal design* and the *messages to be sent*.

Consider design modeling and construction for a vehicle simulation system. The results from analysis are the domain classes *car*, *motor* and *wheels*. During *external design modeling*, the public interfaces of these classes and the messages between them are specified or refined. This is done independently of the selected construction environment, e.g. a design tool is used to draw a diagram.

One of the activities in *internal design modeling* is the refinement of the class *motor*. It has a private attribute *state* (which can be *on* and *off*). This attribute has the default value *off*. The *implementation design modeling* adds an algorithm for the operation *run* to *motor*. To perform this task, new private methods like *startIgnition* and *injectFuel* are defined.

Design construction is used to map these results to the construction environment. If, for example, an ODMG ODBMS is used to construct the system, during *external design construction* the public interfaces are mapped to ODL (Object Definition Language). During *internal design construction*, the attribute *state* is added to the C++ binding generated of the ODL specifi-

cation. During *implementation design construction* the algorithm of the operation `run` is implemented.

Now that we know what different activities constitute the design process, we can look at a specific problem domain: our "design target" is an *objectbase*. In this thesis, we do not consider *design construction* (which would assume that we have a specific construction environment in mind) but *design modeling*. The objective of the design modeling approach proposed here is to help designers achieving the required properties of the conceptual *objectbase schema*.

4.2 Designing a *good* Objectbase

Objectbase design is software design and software design is design.

An objectbase is a logically coherent collection of objects. It contains the state (data) *and* the behavior of objects. Object-oriented databases (Atkinson, Bancilhon et al. 1989; Bancilhon, et. al. 1992; Heuer 1992; Bertino, Marino 1993; Cattell 1994; Barry 1996; Lausen, Vossen 1996) and object-relational databases (Kim 1992; Stonebraker, Moore 1996), CORBA servers (OMG 1995; OMG 1997) and Java RMI (Remote Method Invocation) servers (Arora 1997; Orfali, Harkey 1997) (see Figure 18) are special kinds of objectbases.

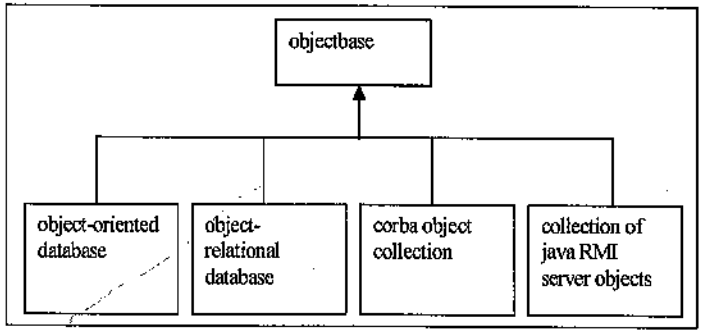


Figure 18: Different kinds of objectbases

Objectbase development is a process with the objective to obtain an objectbase system. An *objectbase system* consists of an objectbase management system and the managed objectbase. An objectbase always has a logical object-oriented schema (sometimes called *objectbase interface*). It describes the objects the objectbase provides to its clients. The description language for this logical schema (objectbase definition language) is provided by the object management system (e.g. ODMG-ODL-, CORBA IDL- and Java RMI Interface).

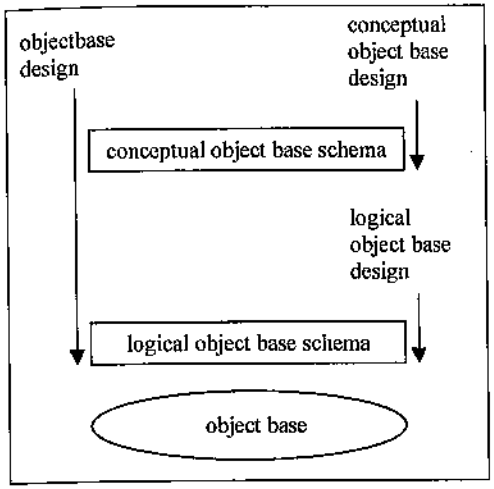


Figure 19: Objectbase design

Objectbase design is a process with the objective to obtain a good objectbase schema.

Objectbase *design* (Figure 19) is object-oriented software design with the objective to obtain a logical objectbase schema. In most cases, it is deferred (sometimes even generated) from a *conceptual* objectbase schema. This conceptual schema can be compared to the conceptual schema of a traditional database, but it contains detailed information about operations, inheritance, composition and other object-oriented concepts.

However, to examine the required properties of such a "good" conceptual objectbase schema, we will recall the key requirements which motivate to choose traditional databases in favor of a file based data management (Elmasri, Navathe 1989):

1. Sharing of data ,
2. adequate representation and common (standardized) understanding of the miniworld ,
3. enforced integrity constraints ,
4. controlled redundancy ,
5. flexibility and
6. reduced application development costs .

These desired features are not achieved automatically, as several years of experience, especially with relational and object-oriented database systems (and with object-orientation in general) have shown:

- neither by any kind of management system (object-oriented, object-relational- database management system (ODBMS, RDBMS), distributed

object management systems (DOMS) like a CORBA or a RMI implementation)
- nor by "pure" usage of object-oriented specification languages.

We have learned that the management systems only provide a toolbox of technical mechanisms and object-orientation only provides a toolbox of expression means that can be used more or less successfully. There is an agreement that object-oriented *software design* is only a prerequisite for successful use of these toolboxes.

To fulfill the characteristics (1)-(6) mentioned above, we propose that an objectbase should fulfill the following requirements:

r1. It should provide effective support for shared objects with explicit control of lifetime and independence of creation and use.
r2. It should contain a comprehensible (standardized) and simple abstraction of (complex) entities of the real world.
r3. The state of the objectbase should be consistent with real world integrity constraints, its behavior should enforce these constraints.
r4. It should be free of anomalies and redundancies.
r5. It should be changeable and extensible with little effort.
r6. It should be independent of other programs, shared and (re-) usable.

In this thesis, we focus on required properties of the *conceptual* objectbase schema. These properties have the strongest influence on the requirements above. A good conceptual schema *enforces* the fulfillment of requirement (r1)-(r6). Such a schema is hard to design and the requirements "shape" the design process. The question is: "How can such a schema be designed?"

The MeTHOOD approach presented here will answer this question. Although the general MeTHOOD framework can be used for software design as well, most of the knowledge in the MeTHOOD catalogues in section 7 and 8 is specifically selected to help fulfilling the requirements (r1)-(r6).

Part II. MeTHOOD

Section 3.4 has shown that integrated design knowledge is missing in current methodologies and section 4.2 has illustrated the specific requirements of objectbase design. These sections are the "driver" for this thesis. This part describes the conceptual frame of MeTHOOD, a design knowledge representation technique allowing integrate design knowledge into an existing methodology. Furthermore, it describes efficient design techniques based on this knowledge representation. Part III presents concrete knowledge catalogues for objetbase design, which help solving the requirements of section 4.2.

5 MeTHOOD concepts

In section 3.4, we have seen that design knowledge is needed in order to support designers fulfilling the core requirements of an objectbase schema. In this section, we present different *concepts* used to *represent* and *integrate* this knowledge in a uniform manner.

Fragments, measures, heuristics and transformation rules are the core of MeTHOOD.

The concepts are *design fragments* (parts of a conceptual design schema), *features* of design fragments and *measures* for these features, *heuristics* allowing evaluation of design fragments and *transformation rules* describing possible improvements.

5.1 Measures and features of a design fragment

Our definitions of measures are based on (Fenton, Whitty et al. 1995). A *measure* in MeTHOOD is an assignment of a number to a design fragment. The terms *metric* and measure are used as synonyms in literature. In mathematics, metrics and measures are different things. A *metric m* is a non-negative, symmetric function of two variables which has a zero and fulfills the condition $m(a,c) <= m(a,b)+m(b,c)$. An example of a metric is the distance between two points. A mathematical *measure* is a non-negative, additive function on a ring of sets, which has a zero for the empty set. Like most proposed metrics, the MeTHOOD measures do not fulfill the properties of mathematical metrics. Therefore, we use the term measure. Measures are used to characterize a specific *feature* of the fragment. A feature of a design fragment is a property influencing its quality, e.g. its change resistance. Combinations of measured values for these features support designers evaluating different design alternatives for a design fragment.

The MeTHOOD measures are subdivided into four families: *size* measures, *hiding* measures, *coupling* measures and *cohesion* measures. (Briand, Morasca et al. 1994) demands that the definition of a measure should be driven by the characteristics of the assumed context in which it is used. These assumptions should be related to a clearly stated goal of the measure. The following section describes the context and the goals of our measures.

Size measures context assumptions (SA):

Our size measures rely on a count of *schema elements* in a design fragment. A schema element is an object of a metatype or a metarelationship (see Table 2 and Table 3), e.g. an attribute, an operation or a parameter.

1. The count of elements in a design fragment may be seen as a measure which is known to be associated with design flaws, i.e. the larger the set of elements, the higher the likelihood of design flaws.
2. The count of elements in a design fragment may be seen as a measure of *understandability*, i.e. the larger the set of declarations, the more effort is required to understand the fragment.

Hiding measures context assumptions (HA):

1. The size of visible fragment elements may be seen as a measure of *size* (from the perspective of the users of the fragment). The larger the visible size of a fragment, the higher the likelihood of design flaws.
2. The size of visible fragment elements may be seen as a measure of *understandability* (from the perspective of the users of the fragment). The larger the visible size of a fragment, the larger the number of external concepts to be used consistently, the more effort is required to understand the fragment.
3. The size of visible fragment elements may be seen as a measure of *consistency*. The larger the visible size of a fragment, the larger the likelihood of a design flaw in an application, which leads to an inconsistency in the fragment.

Coupling measures context assumptions (CA):

1. The stronger a fragment is coupled to others, the more different ways of usage exist, the higher is the likelihood of an inconsistent or wrong use.
2. The stronger a fragment is coupled to others, the more effort is required to understand the different ways of usage of the fragment, the more effort is required to trace back from the changed location to all users (as described in (Weiser 1982)).
3. The stronger a fragment is coupled to others, the more effort is required to find all locations where changes in the fragment lead to subsequent changes out of the fragment.

Cohesion measures context assumptions (COA):

Our cohesion measures rely on our coupling measures. We consider cohesion of a fragment as its *internal coupling*. Please note that this notation is held in the sense of the cohesion definition in (Yourdon, Constantine 1997), where cohesion is considered as "togetherness of a module". It is more common than well-known cohesion definitions, especially LCOM[9] in (Chidamber, Kemerer 1991), where cohesion is measured on the class level.

[9] See Henderson-Sellers, B. (1996). Object-Oriented Metrics: Measures of Complexity, Prentice Hall. p.143 for en extensive discussion of LCOM.

1. The lower the degree of cohesion of a fragment, the more dependencies are scattered in the schema, the higher is the likelihood and the effort for changes in the fragment resulting for other changes outside the fragment.
2. The lower the degree of cohesion of a fragment, the more dependencies are scattered in the schema, the more effort is required to understand the fragment.
3. The lower the degree of cohesion of a fragment, the more dependencies are scattered in the schema, the harder it is to find all locations where a change has to be performed.

These assumptions have driven the definitions of our measures defined in chapter 7. They also give a general rationale for many of the heuristics we propose in chapter 8. Please note, however, that many of the assumptions can be "weakened" by other techniques than the schema inspection/modification proposed here. E.g. (CA2) can be "weakened" by a software development environment that makes dependencies between fragments more transparent and supports automatic change updates.

Please note also that measures can only give an overall understanding of the features of a design fragment. They do not support the detection of a specific structural misfit in a schema, e.g. a redundant relationship, which is often the reason for a "bad" value for a specific measure. MeTHOOD supports designers detecting this kind of misfit by means of *design heuristics*.

5.2 Design heuristics and transformation rules

A *design heuristic* is a rule of thumb. It is an advice for designers on how to use design techniques in order to improve the quality of a design fragment. Sources of heuristics include the literature on object-oriented design, guideline documents of companies and personal experience. A heuristic is not a law. We see a set of design heuristics as a "coloring technique" which marks potential design flaws violating the heuristic.

There are heuristics where it is not objectively decidable whether any design fragment violates them, sometimes it is not even possible to decide if the heuristic can be applied (the situation described in the heuristic occurs in the fragment). Consider for example the heuristic "Distribute the system intelligence as uniformly as possible among your top level classes." (Riel 1996). There are other heuristics where violation can be checked by the designer or by an algorithm. Therefore, we distinguish the following cases:

A *checkable heuristic* can be checked objectively in a given design fragment. This means that for every given design fragment, there is an algorithm

which computes whether the fragment is in conflict with the heuristic or not. More formally, a checkable design heuristic h is a mapping $h: F->\{true, false\}$ where F is the set of all possible design fragments. For a design fragment f, $h(f)$ is *true* if h is not applicable to f or h is fulfilled. $h(f)$ is *false* if h is applicable to f and h is not fulfilled. In other words, if $h(f)$ is *true*, f needs no changes, from the point of view of the heuristic. If $h(f)$ is *false*, f contains a potential design flaw. In this case, we say the heuristic "fires". A firing heuristic marks potential design flaws in a design fragment. Examples of checkable heuristics are "Do not change the state of an object without going through its public interface" or "A class in a containment hierarchy should only depend on its child classes". Checkable heuristics are always achievable, that means a non-trivial design fragment without conflicts exists.

Heuristics are no strong rules.

An *implied heuristic* is an advice to optimize the value of a measure. Consider, for example, the heuristic "Minimize the number of classes a class collaborates with". It is implied by the measure "Number of classes a class collaborates with" (small values of this measure are better than large values). Implied heuristics have no "hard" checking algorithms. Sometimes they are not achievable because there is no "best value" for the corresponding measure (e.g. "Minimize the number of operations in a class.").

Uncheckable design heuristics cannot be checked automatically for a given design fragment. An example of such a rule is "Object-oriented designers should never allow physical design criteria to corrupt their logical designs." This kind of advice is usually very important. It often subsumes and gives rationale for "lower level" checkable heuristics. In most cases, it is a hint towards an ideal design that is not achievable, e.g. a design completely open for extension and completely closed for modification (open/closed principle).

Heuristics are used to *detect* misfits in a conceptual schema but they do not help to *correct* them. A *transformation rule* describes a process that is used to give designers proposals for (restructured) alternative design fragments. These alternative fragments preserve the existing functionality and the semantics of the original fragment but do not contain potential design flaws.

5.3 The MeTHOOD Integration

In the last section, we have seen different knowledge representation techniques used in MeTHOOD (measure, heuristics and transformation rules). In this section, we consider the *integration* of these techniques. This integration is one of the essential concepts of MeTHOOD. It eliminates some of the disadvantages of the considered approaches, and exploits the implicit relationships between these techniques to make design decisions more objective and transparent.

We have observed that designers seek support for reasoning more objectively about a design, finding design flaws, and deciding between various possible design alternatives. Therefore, we have three main requirements for any approach that claims to support the design of a good conceptual schema:

1. It must help designers to evaluate consequences of schema modifications.
2. It must give designers information about possible design flaws.
3. It must provide proposals on how recognized design flaws can be removed.

Current approaches which use measures and design heuristics could be very effective in providing this kind of support. However, they address the above-mentioned requirements only partly.

In these approaches, measures are used at certain well-defined points in the software development cycle to find out how certain projects perform. They are used to estimate (Humphrey 1995), to compare different projects, to predict the development and maintenance costs, to make statements about the quality of the software (for example in terms of defects per module) and to optimize the development process. One of the main disadvantages of current product measures is that they only deliver passive numbers. Using these numbers, the designer knows that something is wrong, and maybe he knows the location. However, there is no systematic approach to making improvements.

The MeTHOOD approach is different because measures are used to give designers *continuous* feedback about design decisions and their consequences (requirement 1). Furthermore, measures are combined with heuristics to find locations where the measured values can be improved (requirement 2). This is different to current approaches that provide "stand alone" heuristics. Another difference to these approaches is that we propose to combine heuristics and transformation rules. This allows proposing useful design fragment transformations to designers (requirement 3). The necessary intégration between measures, heuristics and transformation rules is facilitated by using the MeTHOOD integration schema.

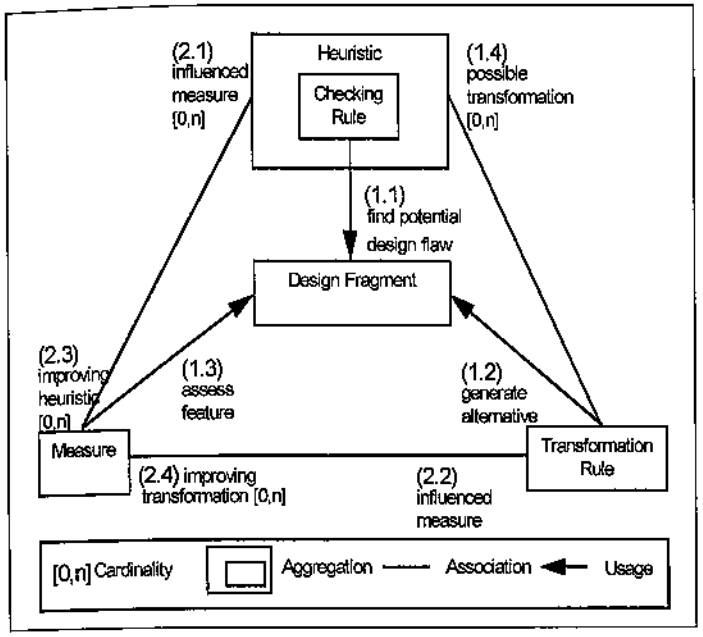


Figure 20: The MeTHOOD integration schema

The MeTHOOD integration schema describes the interesting relationships between measures, heuristics and transformation rules. Consider Figure 20: The MeTHOOD integration: Heuristics are used to find potential design flaws (1.1) in a design fragment, transformation rules are used to generate alternative design fragments (1.2), and measures assess features of the design fragment (1.3).

If a designer has found a potential design flaw (using a firing heuristic *h*), he can decide to remove the violation of *h*. All modifications which remove violations of a certain heuristic *h* change the structure of all design fragments in a similar way. Consider, for example, the heuristic "A class in a containment hierarchy should only depend on its child classes". Removing a violation of this heuristic always means to remove a certain kind of relationship between contained classes and classes different to their child classes. The fact that these structure changes are similar allows describing and formalizing the change process. Therefore, we propose to associate heuristics with transformation rules (1.4) in the MeTHOOD integration schema. The heuristic describes how to detect a design flaw; the transformation rule describes how to eliminate it. More formally, a transformation rule is a mapping *t: F->F* on the set *F* of all design fragments, and a transformation rule

belongs to a checkable or an implied heuristic if it fulfils one of the following transformation criteria:

Checkable heuristic: Let *h* be a checkable heuristic and *f* in *F* a violating design fragment with *h(f)= false*. *t* is a transformation rule for *h* in *f* if *h(t(f))= true*.
Implied heuristic: Let *j* be an implied heuristic and *m: F-> R* the measure of *j*. Let *f* be a fragment in *F*. *t* is a transformation rule for *j* in *f* if *m(t(f))* is better than *m(f)*.

Transformations influence measured values.

Furthermore, since transformation rules change design fragments in a similar way, the specific measured values of the resulting design fragment will also change. These value changes are predictable for many heuristics and transformation rules. Making a violating design fragment compliant with the heuristic "A class in a containment hierarchy should only depend on its child classes" always lowers the effort of isolating this fragment. Corresponding measured values improve. Therefore, the MeTHOOD integration schema also includes associations between heuristics and measures they influence (2.1)/(2.3) and between measures and transformation rules (2.2)/(2.4). This allows selecting heuristics systematically, according to desired improvements of measured values. Finally, it helps designers to solve conflicts between colliding heuristics, one of the main problems of "pure" heuristic approaches.

The integration schema is a technique that allows *representing design knowledge* and making is usable in a systematic manner. By representing the knowledge in this integration schema, designers facilitate efficient new design techniques, as illustrated in the following section.

6 MeTHOOD Design techniques

In the last section, we have seen that MeTHOOD is a methodology *extension*. Its focus is on providing and representing *design knowledge* and the *technical support* for the individual designer to allow a move efficient design and inspection process (Porter, Votta 1997). This following section describes *how* to work with MeTHOOD. It includes various MeTHOOD design *techniques*. All of them are based on the MeTHOOD process.

6.1 The MeTHOOD Process

Iterative design monitoring and improvement

The MeTHOOD process (Figure 21) consists of different activities which are performed iteratively by designers to improve a design fragment.

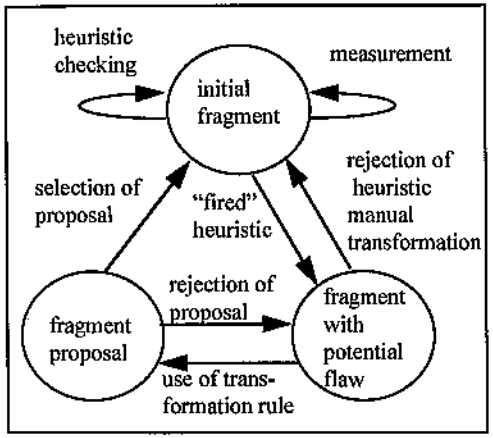


Figure 21: The MeTHOOD process

These activities result in state transitions of the considered fragment. In the *initial fragment* state, the designer can measure values of different features

of the fragment. He may then decide that one of these values has to be improved. In order to find locations in the fragment where improvements are possible, he can check the fragment for improving heuristics of the corresponding measure. If such a heuristic "*fires*", the state of the fragment changes to *fragment with potential design flaw*. This state change also occurs when the designer performs a check on all available heuristics and one of them fires.

Violations are potential flaws, not errors.

If a design fragment is in the state *fragment with potential design flaw*, it contains a violation of a heuristic. The designer has different options of resolving this conflict. One option is to *reject the heuristic*. The rejection disables the heuristic for this fragment. Another option is to *remove the violation manually*. This means, the designer executes a set of schema transformations which remove the violation of the heuristic. Both of these options change the state back to *initial fragment*, where the designer can observe the effects of the provided transformations on the measured values of the fragment. The third option is to check if the violated heuristic has a transformation rule. If this is the case, the transformation rule can be used to *generate a proposal for a fragment* that does not contain violations of the heuristic.

This changes the state of the fragment to *fragment proposal*. In this state, the designer can see how the fragment can be transformed to comply with the violated heuristic. He can *reject* the proposal which leads back to the "*fragment with potential flaw*" state. If he *selects the proposal*, the transformation rule is executed (eliminating the conflict) and the state changes to "*initial fragment*".

6.1.1 Car example

Consider a design fragment of a vehicle simulation system as sketched in Figure 22 (A). The *car* class contains a *motor* class and a *wheel* class.

It has an operation *drive()* which uses the operation *run()* of *motor*. The operation *run()* uses the operation *turn()* of the *wheel* class. In this example, the designer plans to use the contained parts of the class *car* also for other vehicles like airplanes and boats. He should therefore reduce the effort of taking a contained class out and using it in another class. The measures for *coupling tension* and *hardness* can be used to solve this task systematically. They are introduced in section 7, so we use a simplified working definition here: a *coupling-hard* class is one that has few dependencies from other classes but many classes depend from it. A class is under *coupling tension* if it depends from many others and many others depend from it.

In this example, the designer should *enlarge the coupling hardness* and *reduce the coupling tension* of motor (since coupling-hard classes with low coupling tension have few dependencies and are easy to isolate from their context).

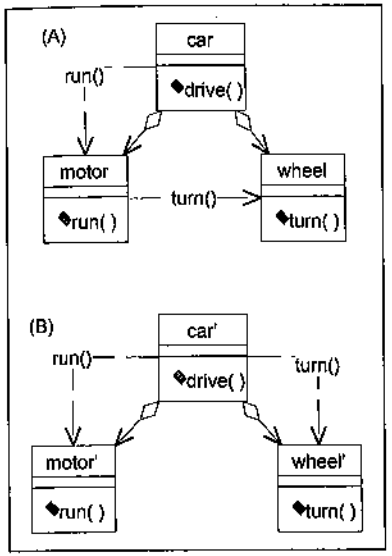


Figure 22: Car example

Table 4: Coupling tension and hardness (A)[10]

	Coupling Tension	Coupling hardness
motor	high	medium
car	low	low
wheel	low	high

Coupling tension and hardness for Figure 22 (A) are illustrated in Table 4. *motor* is not very hard and has a high coupling tension. The designer now checks all heuristics that have a positive effect on these measures. One of the heuristics with a positive effect on coupling tension and hardness is "A class in a containment hierarchy should only depend on its child classes", as defined in section 8.2. A part of the rationale of this heuristic fits to our problem:

"*Reuse of abstraction:* If a contained class depends on its container or its siblings there are two alternative actions to use it in another context:

- it has to be isolated (the dependencies have to be eliminated)
- the depending classes have to be provided in the new context

[10] In the measures catalogue in section 7 we present measures that can be used to calculate coupling tension and hardness of a design fragments.

Both options are not desirable and require an additional development effort. The first option results in code changes and a completely new version of the class. "

The heuristic is violated. A checking rule and a transformation rule are used to remove the design flaw:

Checking rules:

Find all contained classes. These classes have a relationship with the metarelationship *hasParts* or *hasAttributes*. In every contained class, find all relationships to other classes than the immediate child classes, e.g. *hasParts* or *hasAttributes* relationships to a base class or a sibling of the contained class.

```
select x,y, b.from,a
from x,y in type, a,b in relationship
where  a.myMetarelationship. name="sendsMessage"
and    b.myMeterelationship. name="hasParts"
and    a.from=x and y in a.to
and    x in b.to and y in b.to
```

Transformation rules:

Add a new relationship from the container to the contained class and remove the dependency between the contained classes.

```
insert b in b.from, y
remove b from x
```

The checking query of this heuristic detects that *motor* has a uses relationship with *wheels* and both are contained in *car*. The *motor* class can (without modifications) only be (re-) used in vehicles where the motor is used to turn the wheels. The transformation rule in this heuristic proposes to remove the uses relationship from *motor* to *wheels* and insert a new uses relationship between *car* and *wheels*. The result (Figure 22 (B)) is presented to the designer, who can accept or reject it.

Table 5: Coupling tension and hardness (B)

	Coupling tension	Coupling hardness
motor'	low	high
car'	low	low
wheel'	low	high

The coupling tension and hardness of the new fragment (B) is shown in Table 15. The strong coupling tension of *motor* is removed and its coupling hardness is now higher.

6.2 MeTHOOD design inspections

Although MeTHOOD is intended for continuous observation of a schema by the individual designer, the MeTHOOD process can also be used for design inspections. In the framework provided by (Porter, Votta 1997) MeTHOOD can be considered as a *systematic detection technique and technology supporting individual analysis*. (Porter, Votta 1997) makes some important suggestions:

1. The structure of the process input and the analysis techniques are more effective than the inspection process.
2. Even well-prepared meetings may find fewer defects than individuals working alone.
3. The techniques supporting individual performance with respect to flaw detection operations have more influence on effectiveness than do non-technical factors.

These results support the claim that it might be useful to consider MeTHOOD for improving inspection effectiveness. Based on these results, we can recommend users of MeTHOOD, who plan to conduct design reviews, to follow some inspection guidelines.

1. They should put a strong focus on the input provided to MeTHOOD. We predict that the more detailed the provided schema, the more effective will be the inspection.
2. They should conduct individual inspections.

6.3 MeTHOOD support for large schemas

It is very hard to get a good overview of a large schema.

MeTHOOD supports different important tasks of designers. It is constructed to support design and inspection of medium-sized and large schemas. Such a schema typically contains more than 100 classes. The main problem of these schemas is that it is very hard for designers to get an overview of the schema. Furthermore, these models are a severe problem for analysis and design tools. Most of these tools assume that a schema can be shown on a computer screen, which is not true for large models. To support the complexity management of large schemas, the MeTHOOD measures are used to provide a statistical overview of the schema. Furthermore, both heuristics and measures are used to define specific *views* on a schema.

6.3.1 Statistical evaluation

If we are asked to evaluate a large schema, the first issue we are interested in is an overview of its dimension. This is important for the estimation and

planning of the further work on the schema. Some typical information we are interested in are overall numbers and sizes, e.g. the number of classes, the number of operations, attributes, etc.

Statistical evaluation provides an overview of the proportions of a schema.

Besides this, such a statistical evaluation gives us an overview of the schema profile. The schema profile is a description of the proportional relations of concepts in the schema. We are interested in measures of the

- number of operations per class,
- number of attributes per class,
- depth of inheritance tree,
- depth of composition tree,
- etc.

These values are independent from the size of the schema. They allow us to make comparisons and get a feeling for general high level problems of the schema, e.g. too small a quantity of classes are used. If we have a feeling for the dimension and the proportions of the schema, we need to take a closer look at the schema.

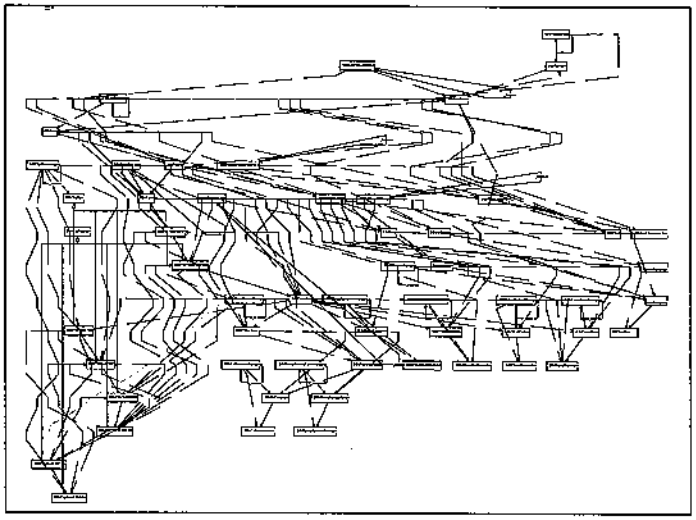


Figure 23: A fragment of a large schema

Views support complexity management of large schemas.

6.3.2 Measure and heuristic based schema views

Finding potential design flaws in a large schema is a serious problem because in general, one does not know where to look.

Figure 23 shows the design of a subsystem of a single banking system[11] (the whole system consists of 23 such subsystems). Attributes and operations have been suppressed because many of the attributes and operations lists would go from the top to the bottom of this page.

To find potential design flaws, it is necessary to examine the schema in detail. This is a very hard task considering the above schema. Using a traditional approach, the designer would now create a printout of a design schema that could cover an office wall.

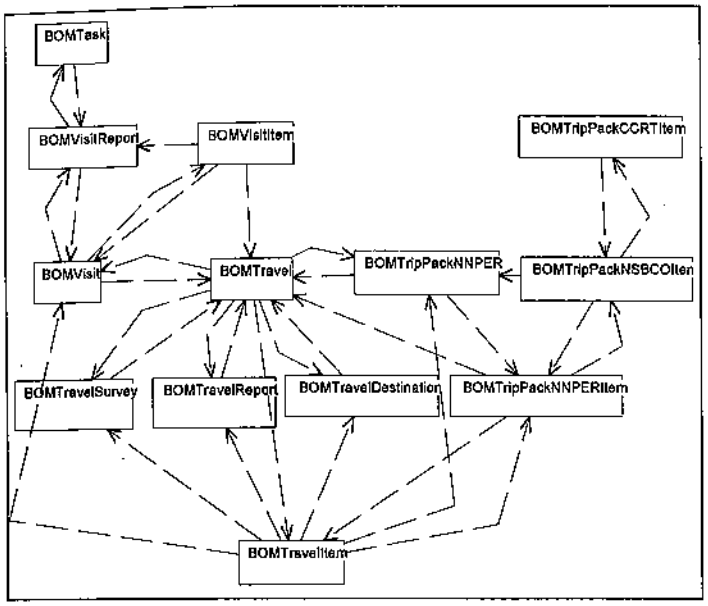


Figure 24: Heuristic based view

Figure 24 shows a *heuristic based schema view* of the schema in Figure 23. Such a view is *the set of all classes participating in a conflict* to a heuristic. In this case, we have a view on all violations of the heuristic: "Avoid dependencies of objectbase classes on their clients" Every class in Figure 24 participates in a client/server relationship, which also includes an undesired relationship between the client and the server. With the view, it is possible to examine the schema further and remove these undesired relationships.

MeTHOOD provides heuristic and measure based schema views.

The example above shows a schema view based on a heuristic. MeTHOOD also provides views based on measures.

[11] This figure is intended to illustrate the dimension and not the contents of the subsystem

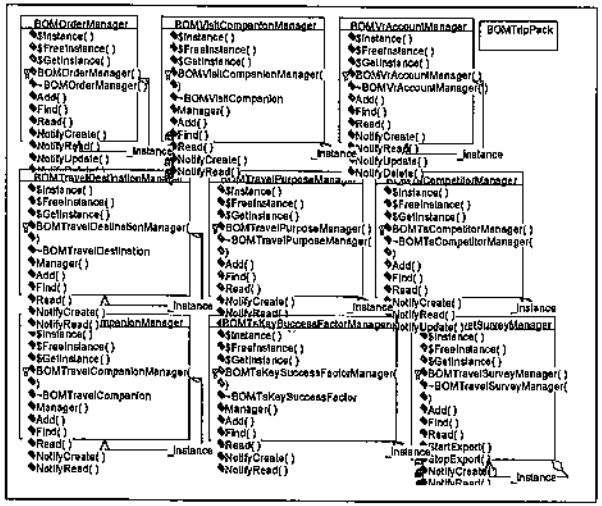


Figure 25: Measurement based filtering

Heuristic and measurement based views show architectural patterns – and anomalies.

Figure 25 shows another view of the same schema. This view contains the classes with the best *ratio between public and private properties* (consider section 7.6 for a detailed description of this MeTHOOD measure). Although this view contains the same types of class relationships as Figure 24, it looks completely different. At first sight, this view is very surprising. The classes have no relationships to each other. Nearly all of them have "Manager" in their names and all of them have nearly the same operation names. If we take a closer look at the architecture of the system, we see that we have in fact filtered all the so-called "Manager-Classes". These classes play a special role in the system. We will consider this example in Chapter 11. It shows that heuristic and measurement views allow designers to focus on special types of classes. In the examined project, this allows to identify cases where classes "break" the architecture – and represent an undesired anomaly.

6.4 Decoupling techniques

Decoupling of classes as modeling activity.

Most of the knowledge captured in the MeTHOOD catalogue considers class coupling. Here, we describe decoupling techniques which support the designer during class decoupling. To consider decoupling more detailed, we examine the following question: Which "coupling information" is usually captured and which part can be considered as essential?

6.4.1 Essential dependencies from the objectbase

Dependencies between client and supplier are a special form of redundancy. The supplier (or server) provides some information and the client (or application) replicates a part of this information. Some of this redundancy cannot be avoided and can be considered as essential. Before we consider techniques of reducing dependencies, we examine which of the information replicated in the application is essentially needed. Consider the following typical code fragment from an application:

```
Customer myCustomer;
Account myAccount:
myCustomer=Customer.select(345);
myAccount=myCustomer.getAccount();
double value= myAccount.getValue();
```

This code fragment allows reconstructing nearly the complete structure of the fragment in the objectbase. The reconstructed structure is shown in Figure 26. All the information needed to construct the schema in this figure is contained in the code fragment.

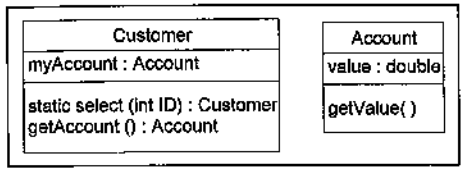


Figure 26: Reconstructed structure

Now consider the following code fragment from the client.

```
double value=Customer.select(345).getAccount().getValue()
```

It achieves the same as the first code fragment. If we consider which information can be reconstructed, we see that this code fragment contains fewer information of the objectbase. It states that a class *customer* with a static operation *select* (which takes an integer) exists (like in the first code fragment). This operation returns an object with an operation *getAccount*(). *getAccount*() returns an object with an operation *getValue*(). We know nothing about the type of this object. We do not know that *Customer* has a relationship to *Account*. We do not even know that a class *Account* exists in the schema!

This demonstrates that the first code fragment redundantly replicates nearly the complete fragment, whereas the second code fragment only replicates a minimal set of required facts. The amount of replication indicates how strong the dependency from client to the schema of the objectbase is. The

weaker the dependencies, the more changes in the objectbase schema are possible; changes that do not affect the applications.

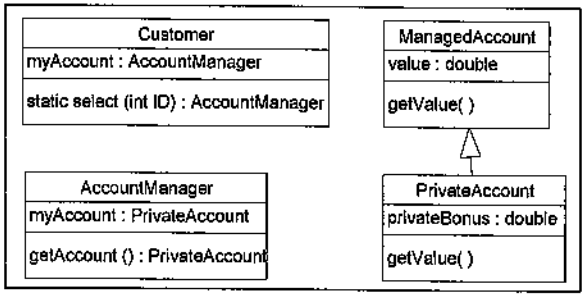


Figure 27: Alternative schema

Figure 27 shows an alternative to the schema in Figure 26. The first code fragment would require massive modifications to be adapted to the schema. The second code fragment does no require a single change.

We conclude that the first step to reduce dependencies from applications to an objectbase is to optimize the application code in order to reflect only the essential dependencies. Eliminating temporary object variables, which are not essentially needed in the application, can do this. The second step is reducing the dependencies between the objectbase classes, using the same technique. After these two steps have been executed, the more "architectural" techniques below should be considered.

6.4.2 Use of mediators to encapsulate dependencies

The mediator pattern is described in (Gamma, Helm et al. 1994). It can be used to *encapsulate* the dependencies from the objectbase in central classes. These classes can be located in the client application or in the objectbase. For example, a mediator can be used to implement the connections between the *Customer* class in the objectbase schema and a customer dialog. A *CustomerDirector* class located in the objectbase can be the single location where state changes of *Customer* objects are managed. The application can create a subclass of *CustomerDirector* called *CustomerDialogDirector* which could manage the communication between the customer dialog and the *Customer* objects. If a state change occurs in the objectbase, *CustomerDialogDirector* would be notified and set the new values in the customer dialog. If the user changes values in the dialog, *CustomerDialogDirector* would be notified and could change the state of the customer object in the objectbase. This example shows that mediators can be used to *encapsulate* dependencies between application and objectbase (the dialog does not know anything about customer objects and vice versa) – the dependencies between the application and the objectbase still exist, but they are defined in another, more central

location. The following sections show how such dependencies can be *eliminated.*

6.4.3 Dynamic data objects

Loose coupling, data dependencies only.

Two classes that are fully decoupled do not rely on their respective structure. In most cases, it is necessary to exchange information between two decoupled classes (otherwise, they would not have been coupled initially). To facilitate this information exchange, a technique called *dynamic data objects* (OMG 1997a) is often used. A dynamic data object is a structured composition of values that is independent from any class. Often, nested attribute-value pairs are used to implement a dynamic data object. Nesting means, the value of attribute-value pair can be another dynamic data object.

Although dynamic data objects allow effective communication (without relying on the structure of the communication partners), they have an inherent disadvantage compared to "native" messages between objects. They have to be created (at the supplier side) and to be transformed into objects at the client side. Both transformations are performance consuming and error prone.

6.4.4 High level application interfaces

A frequently used way to decouple an application from an objectbase and to allow traceability is to define application specific high-level interfaces which can be mapped to applications of the objectbase.
Figure 28 shows a simple example of a high level-interface[12]. Both sections of the example consist of a Customer class and an Account class. The Customer class has a reference to the Account class.

In the lower section of the figure, a low-level interface is used. The *CustomerFactory* class delivers *Customer* objects. The *Customer* objects deliver *Account* objects and the *Account* objects can be asked for the *amount*. In the upper part of the figure, the *CustomerFactory* delivers all information of *Customer*. Basically, the navigation is transferred to the *CustomerFactory* class, only actions can be executed and the clients do not know objects. A prerequisite for this type of communication is a dynamic data protocol (OMG 1997a), which allows composing values to complex "data objects" and returning these objects.

The main advantage of the high level interface is that the application is decoupled from the objectbase, and changes in the objectbase are traceable. Consider, for example, a name change from *Customer* to *AccountOwner*. The decoupled high-level interface can be used to shield this change from all applications. In general, fewer changes in the objectbase schema will propagate to the application. Another advantage is that changes in the objectbase

[12] We use a different notation here to show the dependencies between the classes in a more accurate way.

schema can now be traced to the application. E.g. if we just need to consider the objectbase, we know which applications are affected by the change.

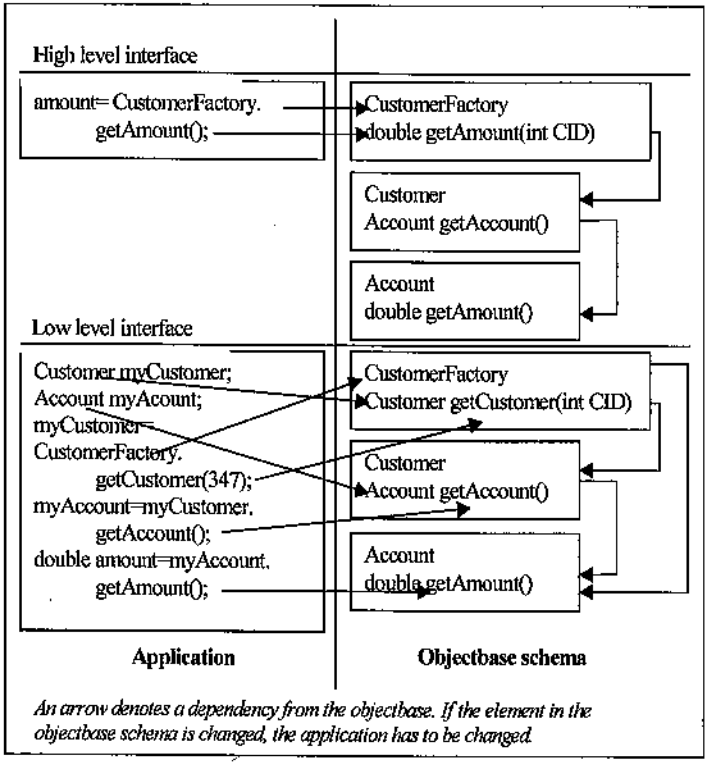


Figure 28: High level objectbase interfaces

Another advantage is that more functionality is available in the objectbase and can be reused by other applications.

This approach has several disadvantages. The communication between objectbase and application *is value based*. Value based communication means that the parameters of the operations provided by the high-level interface and the returned results are values. These values can be quite complex; however, they often have to be transformed to objects within the applications. Finally, using the objectbase is not transparent for programmers using an object-oriented language. It is like using a procedural application interface.

6.4.5 Objectbase bus

Busses: decoupling and robustness.

Another technique to decouple the application from the objectbase is to use a bus architecture that relies on a publish/subscribe mechanism. This is a well-known principle, mainly for object-oriented middleware (e.g. TIBCO's information bus (TIB) (TIBCO 1999a; TIBCO 1999b)) or SoftWired's iBus (SoftWired 1999). The main idea of a bus architecture is to decouple the communication of different objects by agreeing on subjects (see Figure 29).

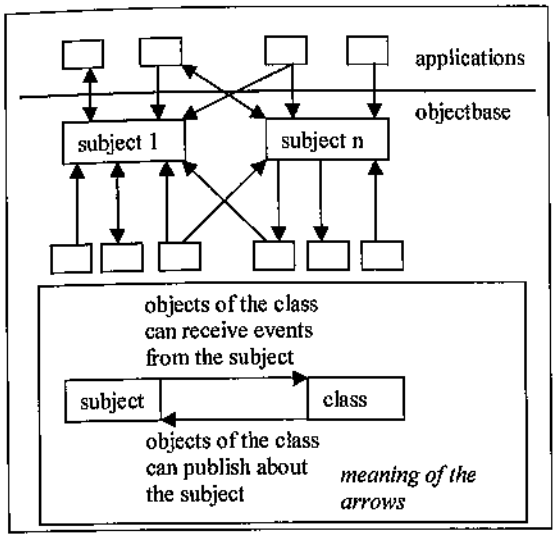


Figure 29: Bus architecture

Objects can publish on subjects, and they can subscribe to subjects.

Figure 30 shows how the example from Figure 28 can be realized using a bus as decoupling technique. Every customer object in the objectbase subscribes to the subject with its identifier. Clients publish their requests on the identifier subject and subscribe to this subject. If a customer object receives a request it publishes the required information on the bus. The publish/subscribe architecture needs an application specific configuration. The designers must decide about the names of the subjects – often a hierarchical namespace is used. Furthermore, one must decide about the format of the message parameter. In the example above we have used a simple attribute value pair to represent a parameter. Again, one can also use dynamic data objects.

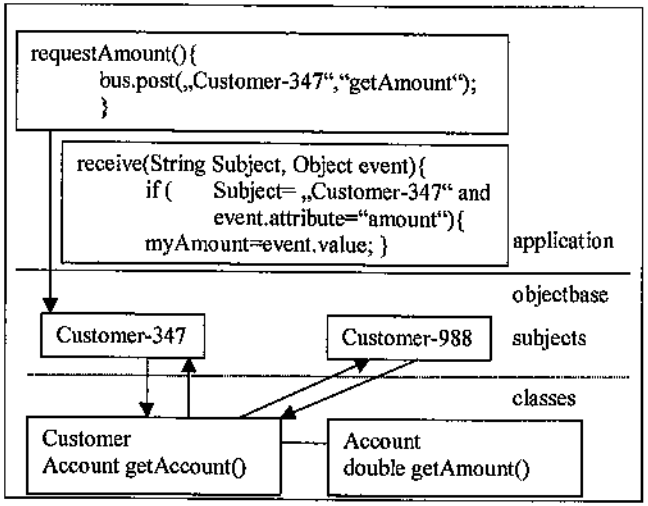


Figure 30: Bus based communication

The main advantage is the full decoupling of the objectbase and the application. Any change in the objectbase that does not consider the subject structure can be performed without effects on the applications. Tracing is now considered on a different level. Changes in the class structure of the objectbase do no longer need to be traceable.

The first disadvantage of this approach is similar to the disadvantage of the high-level interface approach. To achieve full decoupling, the communication must be value based. A further disadvantage is that some of the transparency of the objectbase is lost on the client side. The developer of the application has a very bad overview about what happens in the objectbase if certain subjects are published.

Part III.
The MeTHOOD Catalogues and their Application

In section 4.2, we have seen the core requirements of a good objectbase and that design knowledge is required to fulfill them. In the preceding part, we have seen how to represent this knowledge. This part contains the required design knowledge for objectbase design. It “links” presented design knowledge with the requirements of a good objectbase, and demonstrates how the application of the design knowledge is supported by a design tool.

7 The MeTHOOD measures catalogue

A major difference between a "well developed" science such as physics and some of the less "well developed" sciences such as psychology or sociology is the degree to which things are measured. (Roberts 1979)

The MeTHOOD measures catalogue contains specific measures which consider size and dependencies between classes. The measured values show designers how a design decision influences the size and the dependencies of a fragment. Before illustrating the different measures, we will give a brief overview of the core ideas of the size and the dependency measures.

7.1 The size model

Many class-based size measures (Briand, Morasca et al. 1994; Chidamber, Kemerer 1994; Lorenz, Kidd 1994; Henderson-Sellers 1996) have been published in the last years. Most of these measures have been developed to replace function point and LOC (Lines of Code) measures. Size measures are especially important because they are the base for many other measures. (Hastings 1996) observes that size is required to measure productivity (productivity=effort/size) and defect density (defect density=defects/size).

Most of the size measures are intended to make cost estimates of building and maintaining the system. Furthermore, they assess the likelihood of errors in the program code, the complexity of the system, the project progress and the testability of the code (Zuse 1995).

In (Zuse 1995), the measures proposed here would correspond to the category "experimental validation of best practices". The intention of the proposed measures is to make fine-grained design decisions more transparent. Therefore our measures must "recognize" size differences based on everything the designer can change. They need to be sensitive of small changes. (Bieman, Kang 1995) also demands this sensitivity to evaluate the relationship between cohesion and reuse.

Consider the classes *MixerSmall* and *MixerLarge* in Figure 31. Size measures based on simple attribute and operation count deliver a size of 3 for both classes.

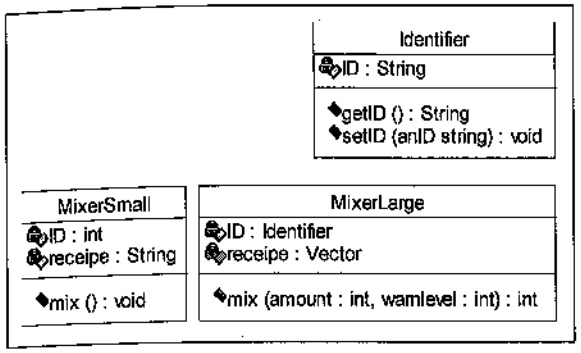


Figure 31: Class sizes

The MeTHOOD class size of *MixerSmall* is 7 and the class size of *MixerLarge* is 12. The MeTHOOD class size considers a class as a composition of properties. The atoms of this composition are the names (of the class, attributes, operations, parameters) and the atomic types (e.g. *boolean, integer, char, string, void*). *Every atom has the size 1.* The values of attributes are added to get values for larger properties. E.g. the attribute ID of *MixerSmall* has the size 2 since it consists of two atoms of size 1.

Our size measure also takes operations, user defined types and the relationships between types into account.

The size of an *operation* is the size of its name (=1) plus the size of the return type plus the size of its parameterlist. The size of the parameterlist of an operation is computed like a set of attributes. It is the sum of the size of all elements in the list. E.g. the parameterlist of mix() in *MixerLarge* is 4. Therefore, the size of the operation mix() in *MixerLarge* is 6.

The size of a *user defined type* is the size of its name (=1) plus the size of all of its properties. We have already seen how properties with simple types are calculated. A property with a complex type (like *ID*) of *MixerLarge* has the size of its name (=1) plus the size of the *interface* of the complex type. We use the size of the interface because only the interface of the type is visible through the property. E.g. in the attribute *ID* of *MixerLarge*, only the operations *getID()* and *SetID()* are visible. The attribute *ID* of *Identifier* is not taken into account. To compute the size of a user defined type we count only the properties in its public interface. Therefore, the size of the attribute *ID* in *MixerLarge* is 3 (*ID*=1, *Identifier*:2).

The size of unary associations is calculated like the size of attributes. If we would replace the attribute *ID* of *MixerLarge* by an unary directed association from *MixerLarge* to Identifier the size of *MixerLarge* would be the same. N-ary associations are calculated like set valued attributes.

7.2 The dependency model

MeTHOOD measures take class relationships and the amount of change propagation implied by these relationships into account. The *dependency model* is a description of the dependencies implied by different types of class relationships. A *dependency* between a client class and a supplier class is a relationship denoting that some changes in the supplier lead to subsequent changes in the client. An *abstract dependency* denotes that some changes in the supplier *can* lead to subsequent changes in the client. Figure 32 shows the different types of relationships that imply dependencies.

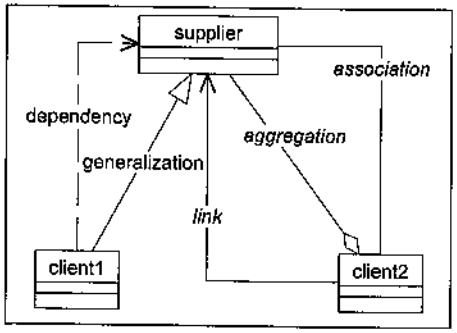


Figure 32: Abstract relationship types that imply dependencies

The dependency model used in MeTHOOD has an *abstract* and a *concrete* form. The abstract dependency model is based on abstract dependencies. Consider the different relationship types in Figure 32. Each of these relationship types denotes a different kind of dependency. These dependencies can be characterized by *the properties of its supplier that can imply the subsequent change*. We call them *critical properties*. A critical property is a property of the supplier of a dependency that can require subsequent modifications in the client if it is modified. The supplier itself (its name) is a critical property for every dependency. Other critical properties of the supplier are certain *attributes and operations*. The client of the dependency uses these elements. If these elements are changed subsequent changes in the clients occur for certain types of changes.

7.2.1 Abstract dependencies

The abstract dependency model uses a "worst case" approach. It simulates a situation in that every *potential* subsequent change in the client of a dependency is a *real* change. Table 6 shows all critical supplier properties for the different types of dependencies. *These properties are taken into account in the dependency measures to measure the amount or strength of the dependency.*

Table 6: Mapping between relationships and dependencies

MeTHOOD metarelationship	UML relationship type	Critical supplier properties
SendsMessage	dependency	public attributes public operations (defined and inherited)
HasBaseClasses	generalization	public attributes public operations protected attributes protected operations (defined and inherited)
Has	link	public attributes public operations (defined and inherited)
HasParts	aggregation	public attributes public operations (defined and inherited)
Has	association	public attributes public operations (defined and inherited)

7.2.2 Concrete dependencies

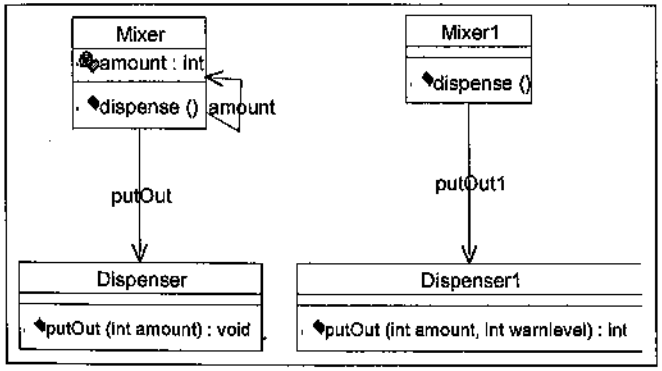


Figure 33: Abstract and concrete Dependencies

A *concrete dependency* is implied by a relationship specified on a deeper level of abstraction. Consider the operation *putOut*() of Figure 33. It is used by dispense() of Mixer. The relationship *putOut*() denotes that Mixer de-

pends on Dispenser (as an abstract relationship). It also denotes that the operation *dispense*() depends on the operation *putOut*() (as a concrete relationship). If we change the name, the parameterlist or the return type of *putOut*(), we have to make subsequent changes in dispense. If we add the parameter *warnlevel*, the code of dispense has to be changed to provide the warnlevel. I.e. if a property used by a client class is changed (added, renamed or deleted), the code of the client class has to be changed subsequently to reflect this change.

The same is true if the operation of the client class uses an attribute of the supplier class. E.g. if the name or the type of the attribute *amount* of *dispenser* is changed or deleted, subsequent changes in the client operation (*dispense()*) occur.

Table 7: A Product Factory class card

Class: Product Factory	Baseclasses
Attributes	
Public Operations Product captureProduct ()	Collaborators Product new Product init
Private Operations	Collaborators
Rules	

SOMA (Graham 1993; Graham 1995) class cards (see Table 7) include concrete dependencies. In the class diagram of UML (OMG 1997b), usually dependencies between classes are specified, and the detailed information cannot be provided[13]. Therefore, MeTHOOD provides an abstract and a concrete mode for measuring. The abstract mode considers UML style class dependencies, the concrete mode considers SOMA class card dependencies and provides more accurate estimates. For precise measurements, the dependencies between two classes must be detailed and concrete. E.g. the specification of depending operations must include all of its suppliers and the used properties of the suppliers.

The MeTHOOD user is responsible for specifying the relationships on *a uniform level of abstraction*. If the user specifies abstract dependencies, MeTHOOD uses measures based on these dependencies. If the user specifies concrete dependencies, MeTHOOD recognizes this, and measures based on concrete dependencies (which provide more detailed values) are used.

[13] However, most case tools provide support for concrete dependencies in class diagrams.

7.2.3 Strength of a dependency

The strength of a dependency depends on the size of the relationship.

Reducing the effort for these subsequent changes is one of the core objectives of design. The effort for the subsequent changes depends on the implementation of the client operation. During class design, the implementation of a class is not known; however, the interface of the supplier class can be used to estimate the effort. To estimate the overall effort, we take into account the probability that a single change occurs in a property and how much effort is needed to do the subsequent changes. Both aspects depend on the *size of the dependent property* (a larger property has more locations where a change can occur, and the set of potential subsequent changes is larger).

Therefore, the dependency model uses the size model. The *strength of a dependency* is an estimate of how much subsequent changes can result from a dependency. The strength of dependencies indicates how much energy is required to execute subsequent changes if a used property of a dependent fragment is changed. *The strength of a dependency is calculated using the accumulated sizes of the dependent properties.*

7.3 Measurement - theoretical foundations

A measure is a mapping from empirical (or intuitively understood) objects to numbers. The mapping should be homomorphic, i.e. the empirical structures and relationships between the measured objects should be preserved. Every proposed measure should be validated empirically and theoretically. The *empirical validation* assures that the measure assesses what it claims to assess. The theoretical validation performed here is based on (Zuse, Bollmann 1989; Fenton 1991; Zuse, Bollmann 1992) (Fenton 1991) (Fenton 1994) and (Kitchenham, Pfleeger et al. 1995). The theoretical evaluation assures that the empirically understood relationships are preserved (i.e. the measure is homomorphic).

In the measures catalogue, we include the field *relation system* (see Figure 34) to validate the measure theoretically. (Kitchenham, Pfleeger et al. 1995) propose two further criteria to be satisfied by direct measures, which are fulfilled for all measures proposed here:

1. To be measurable, an attribute must allow different entities to be distinguished from one another. (Different fragments can be distinguished.)
2. Different entities can have the same attribute value (within limits of measurement error). (Different fragments can have the same attribute value.)

An initial empirical evaluation has been performed and is described in chapter 11.

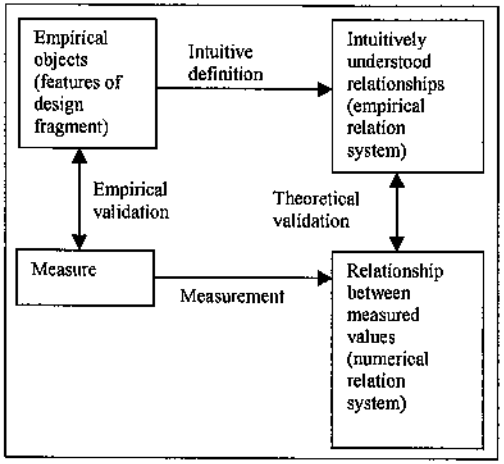


Figure 34: Validation framework

Besides the validation, we have to show which *scale type* the measure has and if it is direct or indirect (Zuse, Bollmann 1989; Fenton 1991; Fenton 1994). The known scale types from the lower to the higher orders are nominal, ordinal, interval, ratio and absolute. *Nominal* scale measures merely assign labels to the measured items, *ordinal* measures provide information about the ordering of the measured items, but the distance between two measured values has no meaning. *Interval* scale measures provide the notion of distance of measured values and have all the properties of ordinal measures. A *ratio scale* adds an absolute zero point to the properties of an interval scale. An *absolute scale* represents a simple count of items.

The scale types determine what interpretations and calculations can be done with the proposed measures, e.g. nominal measures do not allow calculation of medians, ordinal measures do not allow to calculate means, but medians are feasible and ratio measures allow calculations of means. The scale type of the proposed measures is included in the field *scale type*. The field *direct/indirect* measure contains information if the measure is *direct* (directly measured feature of a design fragment) or *indirect* (formula based on other measures).

7.4 Format of the measure description

This section shows how the measures in the catalogue are described:

Description:
Every section starts with a short informal description of the measure.

Motivation:
Why do we measure this attribute? The context assumptions in section 5.1 give an overall answer for the family of measures this measure belongs to. In this section, we give a more specific answer for the proposed measure.

Assessment/prediction:
Do we make an assessment or a prediction? Of what?

Kind (product/process/resource), granularity (class, method):
What is measured?

Direct measure/indirect measure, external attribute/internal attribute:
How do we measure? (see 7.3)

Required level of detail:
What information do we need to calculate the measure?

Atomic modifications:
How sensitive is the measure to an atomic modification (Henderson-Sellers 1996) in the measured fragment?

Formula:
The formula that represents the measure.

Relation System:
The relation system describes which relations on the scale of the measure can be usefully interpreted. The measurements in this thesis are primarily used to compare different fragments and to illustrate the effects of heuristics and transformation rules. Therefore, we focus on the following relations:

- *Equality*: Two fragments are equal if they have the same classes and the same relations between the classes.
- *Comparability*: A < relation on fragments is preserved on the scale of the measure.
- *Additivity*: The operation is based on the union u of fragments. The union of fragment f1 and f2 (f1 u f2) contains all classes of f1 and f2.
- *Multiplicity*: The operation is based on the corresponding definition for real numbers.

Scale type: (nominal, ordinal, interval, ratio, absolute)
(see 7.3)

7.5 Size of design fragment (SF)

SF measures *the size of a design fragment* according to the MeTHOOD size model (see section 7.1). The size of design fragments should be minimized.

Motivation:

Designers need to know how large a design is. They need to know which classes are particularly large (and need to be split) and which classes are small (and can probably be removed). The size measure is especially important to assess how design changes influence the design.

Assessment/prediction:

Assessment of fragment size.

Kind, Granularity:

Product measure of design fragment.

Direct measure/indirect measure, external attribute/internal attribute:

Direct measure.

Required level of detail:

Classes with inheritance, private and public areas, operations with parameters and parameter types, operation return types, attributes with types.

Atomic modifications:

Table 15: Atomic size modifications

Modification	Schema Element	Formula	Expected change of value
Add Element	Schema	S9	Superschema: Increase by 1
	Class	S8	Schema: Increase by 1
	Attribute	S1	Class: Increase by 1
	Operation (inherited)	S3 (S8)	Class: Increase by 1
	Operation (new)	S3	Class: Increase by 1
	Message	S5	Operation: Increase by 1 + size of operation
	HasParameter	S1	Class: Increase by 1
	Has, HasParts	S2	Class: Increase by size of public interface of supplier
	HasBaseClasses	S8	Class: Increase by size of public and protected interface of supplier

Modification	Schema Element	Formula	Expected change of value
Remove Element	Schema	S9	Superschema: Decrease by 1
	Class	S8	Schema: Decrease by 1
	Attribute	S1	Class: Decrease by 1
	Operation (inherited)	S3 (S8)	Class: Decrease by 1
	Operation (new)	S3	Class: Decrease by 1
	Message	S5	Operation: Decrease by 1 + size of operation
	HasParameter	S1	Class: Decrease by 1
	Has, HasParts	S2	Class: Decrease by size of public interface of supplier
	HasBaseClasses	S8	Class: Decrease by size of public and protected interface of supplier
Change Name	All elements	S1-S9	No size changes
Add Type	AttributeType (new and inherited attributes)	S6, S7 (S8)	Void type: No size changes Atomic type: Increase element by 1 Set, list, tuple type: Increase element by 1+content type Tupletype: Increase element by 1 + numver of attributes Classtype: Increase by 1 + size of public interface of supplier class
	ReturnType (new and inherited operations)		
	ParameterType (new and inherited operations)		

Formula:

S1: **size of attribute = size of operation parameter =**
1 + size of type

S2: **size of association to class B =**
unary ((0,1)(1,1)) association = 1+size of type of class B //corresponds to attribute
n-ary((0,n)(1,n)) association = 2 + size of type of class B //corresponds to set type. The size of the set type is 1+ size of content type. E.g size(association to Customer) = 1+size(Customer) and size(association to set(Customer)) = 1+ size(set(Customer)) = 1+1+size(Customer).

S3: **size of operation without collaboration =**
1 + sum of size of all parameters + size of return type

S4: size of operation with collaboration =

1 + sum of size of all parameters + size of return type + size of every message = size of operation without collaboration + size of every message

S5: size of message =

1 + size of receiving operation without collaboration

S6: size of type =

void type:	0
atomic type:	1 (integer, char, enum, real, ...)
set/list/array type:	1+ content type
tupletype:	1 + sum of size of all tuple attributes
classes:	size of class interface'

S7: size of public class interface =

1+
size of public operations without collaboration +
size of public inherited operations without collaboration +
size of public attributes + size of public inherited attributes +
size of public associations + size of public inherited associations
(The size of the public class interface contains everything that is provided by this class)

S8: size of class =

1 + size of attributes (including inherited) + size of associations (including inherited)+ size of new operations with collaboration + size of inherited messages without collaboration

S9: size of fragment SF=

1+sum of all fragments + sum of size of all classes within fragment

Examples: **size**

```
class Car{                                  1
private:
        int colour;                         2
        Wheels myWheels;                    3
        Motor myMotor;                      3
public:
        boolean start()                     2
                myMotor.start();
                myWheels.turn();
}

SF({car}) =                                 11
```



```
class Motor{                                1
public:
        boolean start();                    2
}

SF({motor}) =                               3

class Wheels{                               1
public:
        boolean turn();                     2
}

SF({wheels})=                               3
SF({car, motor, wheels})=                   17
```

Example: (Handling of dependent classes)

```
class Wife{                                 1
public:
        Husband aHusband;                   3
}

SF({Wife})=                                 4

class Husband {                             1
public:
        int age;                            2
        Wife aWife;                         2
}

SF({Husband})=                              5
SF({Wife, Husband})=                        9
```

Example: (Handling of complex types) **size**

```
class Car{                                  1
        set of tuple<int size, int type> wheels;    7
}
                        set of              1
                      + tuple               1
                      + int size            2
                      + int type            2
                      + wheels              1
SF({car})=8;
```

Relation System:
Let size(f) be the intuitively perceived size of a fragment f:

- *Equality:* size(f1)=size(f2) <-> SF(f1)=SF(f2)
- *Comparability*: size(f1)<size(f2) <-> SF(f1)<SF(f2
- *Additivity*: SF(f1 u f2)=SF(f1)+SF(f2)
- *Multiplicity*: size(f1) = n*size(f2) <-> SF(f1) = n*SF(f2).

All statements above are plausible from the definition of SF. SF is sensitive to every change in a fragment because it assesses all perceived elements of the fragments; therefore, equality and comparability are plausible. According to the definition of SF and the union of two fragments, additivity is given. The definition of SF directly indicates that multiplicity is plausible.

Scale type:
The empty fragment has a size of 0 (SF(e)=0). This measure has a ratio scale. The values range from 0 to N where N is a positive integer.

7.6 Public factor of design fragment (PF)

The *public factor* is the probability that a property of a design fragment is public available. The public factor of design fragments should be *minimized.*

Motivation:
Information hiding is one of the most important principles that should be followed in object-oriented design. This measure assesses the degree in which properties are encapsulated in a design fragment. It is similar to the "method[14] hiding factor (MHF)" and "attribute hiding factor (AHF)" proposed in (Abreu, Melo 1996).

Assessment/prediction:
Assessment

Kind, Granularity: Description of measure.
Product measure of design fragment.

Direct measure/indirect measure, external attribute/internal attribute:
Indirect measure.

Required level of detail:

[14] Operation hiding factor in MeTHOOD terminology

Classes with inheritance, private and public areas, operations with parameters and parameter types, operation return types, attributes with types.

Atomic modifications:

Table 8: Atomic public factor relations

Modification	Schema Element	Expected change of value
Add Element	public element: attribute, association, operation (including public inherited)	Increases measured value
	private element: attribute, association, operation (including protected inherited)	Decreases measured value
Remove Element	public element: attribute, association, operation (including public inherited)	Decreases measured value
	private element: attribute, association, operation (including protected inherited)	Increases measured value

Formula:
If a class has no properties the public factor is 0.

public factor

= 0 for the empty fragment e
= (sum of size of every public property (including public inherited) +
sum of size of every public association (including public inherited))/
(sum of size of all properties (including all inherited) +
sum of size of all associations (including all inherited)) otherwise

Examples:

```
class Car{                                  1
private:
        int colour;                         2
        Wheels myWheels;                    2
        Motor myMotor;                      2
public:
        boolean start()                     2
                myMotor.start();
                myWheels.turn();
}

PF({car}) =(1+2)/(1+8)                      0.333
```

```
class Motor{                                        1
public:
        boolean start();                            2
}

PF({motor}) = 3/3 = 1

class Wheels{                                       1
public:
        boolean turn();                             2
}

PF({wheels})= 3/3 = 1

PF({car, motor, wheels})=(3+ 2+2+2)/(3+ 8+2+2)=0.6
```

Relation System:
Let public(f) be the intuitively perceived amount of unhidden elements of fragment f:

- *Equality*: public(f1)=public(f2) <-> PF(f1)=PF(f2)
- *Comparability*: public(f1)<public(f2) <-> PF(f1)<PF(f2)
- *Multiplicity*: public(f1)=n * public(f2) <-> PF(f1) = n * PF(f2)

Equality, comparability and multiplicity are directly plausible form the definition of PF. This measure is not additive.

Scale type:
The empty fragment has a public factor of 0 (PF(e)=0). The measure has a ratio scale from 0 to 1. The distance between two measured values indicates the percentile difference of the share of hidden elements in the measured items. E.g., if *PF(f1)=0.5* and *PF(f2)=0.33* the share of public elements in *f1* is 50%, and the share of public elements in *f2* is 33%. The share of public elements of *f2* is 17% larger than in *f1*. Large values indicate poor hiding, small values indicate good hiding.

7.7 Coupling of fragment (CF)

CF assesses how strong a source fragment depends on classes in a target fragment.

Motivation:
Fragments with few dependencies between them characterize a modular design. Excessive dependencies between fragments prevent their reuse.

Furthermore, the more dependencies a fragment has to classes outside the fragment, the higher is its sensitive to changes outside the fragment.

Assessment/prediction:
Assessment

Kind, Granularity:
Product measure of design fragment.

Direct measure/indirect measure, external attribute/internal attribute:
Direct measure

Required level of detail:
Classes with inheritance, private and public areas, operations with parameters and parameter types, operation return types, attributes with types.

Atomic modifications:

Table 9: Atomic coupling modifications

Modification	Schema Element	Expected change of value
Add Element to class in source fragment	message to target fragment	Increases measured value by size of message
	HasBaseClasses to target fragment	Increases measured value by size of inherited properties
	association to target fragment (including attribute association)	Increases measured value by size of the public interface of the supplier
Remove Element	message to target fragment	Decreases measured value by size of message
	HasBaseClasses to target fragment	Decreases measured value by size of inherited properties
	association to target fragment (including attribute association)	Decreases measured value by size of the public interface of the supplier

Formula:

Coupling of fragment(source fragment, target fragment)=
Sum of size of all messages +
Sum of size of all associations (including attribute associations) +
Sum of size of all inherited properties (source fragment inherits from target)

from classes inside the source fragment to classes inside the target fragment.

Often, the target fragment is the whole schema. For convenience, we also introduce coupling where only the source fragment is given:

Coupling of fragment(source fragment)=
Coupling of fragment(source fragment, target fragment)
where target fragment is the whole schema with the source fragment removed.

Examples: **size**

```
class Car{
private:
        float speed;
        Wheels myWheels;                5
        Motor myMotor;                  3
public:
        boolean start()                 2
                myMotor.start();        3
                myWheels.turn(s);       5
}

class Motor{
public:
        boolean start();                2
}

class Wheels{
public:
        boolean turn(int s);            4
}
```

CF({car}, {motor})=size(myMotor.start())+size(Motor myMotor)= 3+3=6
CF({car, wheels}, {motor})=6
CF({car})=CF({car},{motor,wheels}) =
size(myMotor.start()) + size(myWheels.turn()) + size(Wheels myWheels) +
size(Motor myMotor) =
3+5+5+3 =16

Relation System:
Let coupling(f) be the intuitively perceived coupling of fragment f:

- *Equality*: coupling(f1)=coupling(f2) <-> CF(f1)=CF(f2)
- *Comparability*: coupling(f1)< coupling(f2) <-> CF(f1)<CF(f2)
- *Additivity*: CF(f1 u f2)=CF(f1)+CF(f2) is valid for fragments f1, f2 without relations to each other. Otherwise, the formula CF(f1 u f2) = CF(f1)+CF(f2)-(CF(f1, f2)+CF(f2, f1)) is valid.
- *Multiplicity*: coupling(f1)=n*coupling(f2) <-> SF(f1) = n*SF(f2)

Equality and comparability are plausible from the definition of CF. The definition of CF and the union of two fragments indicate the additivity. For multiplicity, we assume that the different kinds of dependencies (associations, inheritance, method calls, etc.) have the same strengths and the same influence on the measured values. This assumption is justified because the coupling measure is based on the size measure. This size measure *normalizes* the different kinds of dependencies because it evaluates all dependencies based on a common denominator: the amount of dependencies to names in the target fragment.

Scale type:
The empty fragment has a coupling of 0 (CF(e, f)=0 for all fragments f). The measure has a ratio scale. Values range from 0 to n where small values indicate low coupling and high values indicate high coupling.

7.8 Inverse coupling of a fragment (ICF)

ICF assesses how strong target fragments depend on a source fragment.

Motivation:
Designers need to know how classes outside a fragment depend on this fragment. This knowledge allows evaluating how likely changes in a fragment result in subsequent changes outside the fragment. Fragments with many dependent classes are hard to change.

Assessment/prediction:
Assessment

Kind, Granularity:
Product measure of design fragment.

Direct measure/indirect measure, external attribute/internal attribute:
Indirect measure

Required level of detail:
Classes with inheritance, private and public areas, operations with parameters and parameter types, operation return types, attributes with types.

Atomic modifications:
The atomic modifications are the atomic modifications of the measure in section 7.7. Inverted means that source and target fragments are interchanged.

Formula:

Inverse coupling for a fragment (source fragment, target fragment) =
Sum of size of all messages from classes in the target fragment to classes in the source fragment.
ICF(source fragment, target fragment) = CF(target fragment, source fragment)

Inverse coupling for a fragment (source fragment) =
Sum of size of all messages from classes outside the source fragment to classes in the source fragment.
ICF(source fragment) = CF(target fragment, source fragment), where target fragment is the whole schema with the source fragment removed.

Examples: **size**

```
class Car{
private:
        speed s;
        Wheels myWheels;
        Motor myMotor;                  3
public:
        boolean start()                 2
                myMotor.start();        3
                myWheels.turn(s);
}

class Motor{
public:
        boolean start();                2
}

class Wheels{
public:
        boolean turn(int s);            4
}

ICF({car}, {motor}) = CF({motor},{car}) = 0
ICF({car}) = CF({motor, wheels}, {car}) = 0 + 0 =0
ICF({motor}) = CF({car, wheels}, {motor}) =3 + 3 + 0 = 6
```

Relation System:
See 7.7.

Scale type:
Ratio scale. The values range from 0 to n, where small values indicate low inverse coupling and high values indicate high inverse coupling.

7.9 Coupling hardness of a fragment (HF)

HF is a factor to determine the change probability of a fragment based on its coupling with other fragments. A fragment with high HF (called coupling-hard or simply hard fragment) is difficult to change because other fragments heavily depend on it. In such a fragment, changes are unlikely because it does not depend heavily on other fragments that could propagate changes. A soft fragment can be changed easily because few fragments depend on it and changes are likely because the fragment depends heavily on others.

Motivation:
Changes in classes outside a given fragment can lead to changes in it. Changes in a fragment can lead to subsequent changes to dependent fragments outside. Dependencies on other fragments are sources of potential change. HF assesses the likelihood of changes in a fragment based on

- the probability of change propagation into the fragment and
- the costs of fragment changes (based on change propagation to others).

HF is an important indicator for the *stability* of a fragment. We define *stability of a fragment* as the probability that the fragment is not modified. Designers need to know if a fragment is stable or instable. In a good design, stable fragments do not depend on instable ones. Other fragments can depend on stable fragments without much danger that changes occur.

Assessment/prediction:
Assessment

Kind, Granularity:
Product measure of design fragment.

Direct measure/indirect measure, external attribute/internal attribute:
Indirect measure

Required level of detail:
Classes with inheritance, private and public areas, operations with parameters and parameter types, operation return types, attributes with types.

Atomic modifications:

Table 10: Empirical HF relation system

Modification	Schema Element	Expected change of value
Add Element to class in source fragment	message to target fragment	Increases measured value
	HasBaseClasses to target fragment	Increases measured value
	association to target fragment (including attribute association)	Increases measured value
Add Element to class in target fragment	Message to source fragment	Decreases measured value
	HasBaseClasses to source fragment	Decreases measured value
	association to source fragment (including attribute association)	Decreases measured value
Remove Element from source fragment	message to target fragment	Decreases measured value
	HasBaseClasses to target fragment	Decreases measured value
	association to target fragment (including attribute association)	Decreases measured value
Remove Element from target fragment	Message to source fragment	Increases measured value
	HasBaseClasses to source fragment	Increases measured value
	association to source fragment (including attribute association)	Increases measured value

Formula:

Coupling hardness of fragment s in a fragment t:
HF(s,t) = ICF(s,t) - CF(s,t)

Coupling hardness of fragment in whole schema:
HF(f) = ICF(f) - CF(f)

Examples:


```
class Car{
private:
        speed s;
        Wheels myWheels;
        Motor myMotor;
public:
        boolean start()                 2
                myMotor.start();
                myWheels.turn(s);
}

class Motor{
public:
        boolean start();                2
}

class Wheels{
public:
        boolean turn(int s);            4
}

HF({car})=ICF({car})/-CF({car})0-16=-16
HF({motor})= ICF({motor}- CF({motor})=6-0=6
```

Numerical relation system:
The numerical relation system is directly implied by the formula.

Relation System:
Let d-stability(f) be the intuitively perceived dependency based stability of fragment f:

- *Equality*: d-stability(f1) = d-stability(f2) <-> HF(f1)=HF(f2)
- *Comparability*: d-stability(f1)< d-stability(f2) <-> HF(f1)<HF(f2)
- *Additivity*: HF(f1 u f2) = HF(f1)+HF(f2) is valid for fragments f1, f2 without relations to each other. Otherwise, the formula HF(f1 u f2) = HF(f1)+HF(f2)-(HF(f1, f2)+HF(f2, f1)) is valid.
- *Multiplicity*: d-stability(f1) = n*d-stability(f2) <-> HF(f1) = n*HF(f2)

The above statements are plausible from the definition of HF.

Scale type:
HF(f) is 0 if CF(f) = ICF(f). The fragments with CF(f) = ICF(f) = 0 represents the absolute 0. The measure has a ratio scale where negative values denote coupling-soft and positive values denote coupling-hard fragments.

7.10 Coupling tension of a fragment (TF)

TF assesses how strong forces (resulting from *dependencies* in different directions) influence a fragment. The coupling tension of fragments should be minimized.

Motivation:

An object is under tension if forces with different directions are applied to it. If other fragments depend on a fragment and the fragment itself depends on other fragments, the fragment is under coupling tension. The dependencies from other classes make it coupling-softer, and the classes it depends upon make it coupling-harder. The result of a high coupling tension is that the probability of changes is high (because it depends on other fragments) and the effort to execute the changes is also high (because other fragments depend on it). High change probability requires a changeable (flexible) fragment. Much dependency on a fragment makes this fragment hard to change because the changes have to be propagated. This situation is not desired.

Assessment/prediction:

Assessment

Kind, Granularity:

Product measure of design fragment.

Direct measure/indirect measure, external attribute/internal attribute:

Direct measure

Required level of detail:

Classes with inheritance, private and public areas, operations with parameters and parameter types, operation return types, attributes with types.

Atomic modifications:

The atomic modifications can be derived from those of section 7.7.

Formula:

The idea behind the coupling tension measure is that coupling and inverse coupling "pull" a fragment f with the forces CF(F) and ICF(F) in opposite directions. The fragment itself is movable. This means, it moves in the direction of the stronger force and rests if the forces are balanced. The force that is now applied to the fragment is its coupling tension. The coupling tension of those forces implied on a *movable* fragment is 2*min(ICF(f), CF(f))[15].

[15] Note: A system of objects and forces that is not movable would require a different formula to calculate coupling tension. We can focus on a movable system because the rate of movement can be calculated using the coupling hardness measure. The higher or lower coupling hardness, the more the fragment has "moved" in one or the other direction. The coupling hardness and tension measures put together provide the whole picture.

Coupling tension of fragment t in a fragment s:

TF(t,s)=2*min(ICF(t,s), CF(t,s))=
2*min(CF(s,t), CF(t,s))

Coupling tension of fragment f in the whole schema:

TF(f)=2*min(ICF(f), CF(f))

Examples:

```
class Car{
        Motor myMotor;
}

class Motor{
        Wheels myWheels;
}

class Wheels{
}

TF({Car}) =0
TF({Wheels}) =0
TF({Motor})=2*min(ICF({Motor}), CF({Motor}))=2*min(2,2)=4
```

Relation System:

Let d-tension(f) be the intuitively perceived dependency based tension of fragment f:

- *Equality*: d-tension(f1)=d-tension(f2) <-> TF(f1)=TF(f2)
- *Comparability*: d-tension(f1) < d-tension(f2) <-> TF(f1)<TF(f2)
- *Additivity*: TF(f1 u f2)=TF(f1)+TF(f2) is valid for fragments f1, f2 without relations to each other. Otherwise the formula TF(f1 u f2) = TF(f1)+TF(f2)-(TF(f1, f2)+TF(f2, f1)) is valid.
- *Multiplicity*: d-tension(f1) = n*tension(f2) <-> TF(f1) = n*TF(f2)

For equality, comparability and additivity, we make the assumption that we deal with a movable system of objects and forces applied to them. Under this assumption equality, comparability, additivity and multiplicity are plausible.

Scale type:

The empty fragment has a coupling tension of 0. The measure has a ratio scale from 0 to n, where 0 denotes low coupling tension and high values denotes high coupling tension

7.11 Cohesion of a fragment (COF)

Cohesion captures the extent to which the properties of a fragment (e.g. operations of a class, classes of a fragment) are conceptually related to the same fragment.

Motivation:
Low value of cohesion in a class is an indication that different concepts are modeled. A fragment consisting of a single class should always be cohesive; a low cohesive class is a candidate for splitting. Design fragments that represent modules or subject areas should be cohesive. Low cohesion increases complexity (Chidamber, Kemerer 1994).

Assessment/prediction:
Assessment

Kind, Granularity:
Product measure of design fragment.

Direct measure/indirect measure, external attribute/internal attribute:
Direct measure

Required level of detail:
Classes with inheritance, private and public areas, operations with parameters and parameter types, operation return types, attributes with types.

Atomic modifications:
The atomic modifications can be derived from those of section 7.7.

Formula:
COF(f) = 0 for the empty fragment
CF(f,f)/|f| where |f| is the number of classes in f

Examples:

```
class Car{
        Motor myMotor;
}

class Motor{
        Wheels myWheels;
}
```



```
class Wheels{
}

class Driver{
}

COF({Car}) =0/1=0
COF({Motor}) =0/1=0
COF({Car, Motor})=
        CF(({Car, Motor }, ({Car, Motor })/2=3/2=1.5
COF({Car, Motor, Driver})=
        CF(({Car, Motor,Driver},({Car, Motor,Driver })/3=3/3=1
COF({Car, Motor, Wheels })=
        CF({Car, Motor, Wheels })/3=5/3=1.66
COF({Car, Motor, Wheels, Driver })=
        CF({Car, Motor, Wheels, Driver })/4=5/5=1.25
```

Relation System:
Let cohesion(f) be the intuitively perceived cohesion of fragment f:

- *Equality*: cohesion(f1) = cohesion(f2) <-> COF(f1)=COF(f2)
- *Comparability*: cohesion(f1) < cohesion(f2) <-> COF(f1) < COF(f2)
- *Additivity*: COF(f1 u f2)=COF(f1)+COF(f2) is valid for fragments f1, f2 without relations to each other. Otherwise, the formula COF(f1 u f2) = COF(f1)+COF(f2)-(COF(f1, f2)+COF(f2, f1)) is valid.
- *Multiplicity*: cohesion(f1) = n * cohesion(f2) <-> COF(f1) = n*COF(f2)

All properties are plausible from the definition of the formula and the corresponding properties of CF (see section 7.7).

Scale type:
Ratio scale from 0 to n. Low values indicate low cohesion, high values indicate high cohesion.

8 The MeTHOOD heuristics catalogue

Heuristics for objectbase design.

The MeTHOOD heuristics catalogue is a collection of over twenty heuristics. We propose these heuristics as rules of thumb to improve the conceptual design of objectbases. Often, these improvements can be measured. In fact, heuristics can detect locations in a schema where measured values can be improved.

8.1 Description of Heuristic format

Every heuristic is described in a consistent format. At this point, we illustrate the meanings of the sections of this format.

The first section below the title *describes* the essence of the heuristics. It answers the questions "What has to be done or what is forbidden to comply with this heuristic?"

Definitions:
Many heuristics include terms specific for the heuristic. These terms are defined in this section.

Rationale:
This is one of the most important sections in a heuristic definition. It describes why a design should comply with the heuristic. It also describes different known contexts in which a heuristic should not be violated.

Position in Life Cycle:
Some heuristics give advice how special concepts of a design fragment should be implemented in a programming language. Others locate missing abstractions during analysis. Most heuristics presented here consider relationship optimizations for design. This section defines the intended position in the life cycle of the heuristic. We frequently use the terms *Analysis, Design and Coding* to describe this position.

Granularity:
Some heuristics can be used to detect anomalies in the dependency relations of a set of classes, others consider operation signatures and operation return types. Every heuristic has such a set of *required language elements*. The examined design fragment must be on a level of detail that includes representation for all of these language elements. This section describes which language elements are "used" by the heuristic. It contains information about which language elements (e.g. class and method) are affected and what kinds of language elements are checked in order to find a violation.

Example:
The example section includes at least two very simple design fragments. In the first fragment, called "Violating", a violation of the heuristic is shown. In the second fragment, called "Complying", this violation is removed.

Subsuming/subsumed heuristic:
Some advice is on a higher level than others. This is also true for heuristics. Let us assume we have two heuristics, A and B. If every design that complies to/violates A thereby also complies to/violates B we only need to check against A. In this case we call A subsuming heuristic and B subsumed heuristic. For every heuristic, we include its subsuming and subsumed heuristics.

Checking rules:
The occurrence of a violation can be found automatically for many heuristics. For heuristics where this is possible we include a checking rule. A checking rule is a description of the process that has to be executed in order to find all violations in a given fragment. A checking rule has to answer the question "How do I check to find all violations of this heuristic?" Some heuristics have different checking rules which consider different aspects of the heuristic. This action includes all known checking rules of the heuristic.

Transformation rules:
Often, it is possible to propose a schema transformation if a heuristic has found a potential design flaw. In most cases, several different schema transformations which eliminate a design flaw are possible. We use transformation rules to describe these transformations. A transformation rule is a description of a process, which can be executed to transform a design fragment with a design flaw into a design fragment without this flaw. This section describes all known transformation rules of the heuristic.

Violated heuristics:
In many cases, heuristics give conflicting advice. One heuristic demands that a design fragment must be in a specific form, whereas the other heuristic finds a potential design flaw in this fragment. This section contains references to all heuristics which are in conflict to the given heuristic.

Justifications for violating the heuristic:

Heuristics are no strong rules. The heuristics presented here describe fragments which represent design flaws in *most* (but not all) cases. Designers should have a good reason to violate a heuristic. Known reasons that justify a violation of a heuristic are presented in this section.

Effects on measures:

Some heuristics require that some elements are removed from the schema. Others propose introducing new classes. The "schema change requests" from a heuristic result in similar changes of the schema. Often these similar changes result in similar changes of measured values of the schema. If we observe that a heuristic always implies a change of a measured value in a certain direction we include a reference to the measure. We note which measures are positively/negatively affected if the violating design is transformed into a complying design.

8.2 A class in a containment hierarchy should only depend on its child classes

A class should neither depend on its container nor on one of its siblings. A class should not depend on classes below its siblings.

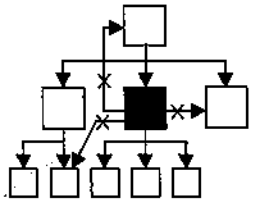


Figure 35: A contained class should depend on its children only

Definitions:

Class A *depends on* class B

- if there is a *hasParts* relationship from B to A, or A is the type of an attribute of B or
- if there is a *using* relationship between an operation in A and an operation in B

Rationale:

Reuse of abstraction: If a contained class depends on its container or its siblings there are two alternative actions to use it in another context:

- it has to be isolated (the dependencies have to be eliminated)
- the depending classes have to be provided in the new context

Both options are not desirable and require an additional development effort. The first option results in code changes and a completely new version of the class.

Complexity reduction: In schemas in which this heuristic is not violated, the containment relationship limits the visibility of every contained class to its parents. Therefore, if a child is changed, only the parents in the containment hierarchy have to be checked for subsequent changes. This is the more important if the schema is an objectbase schema with many dependent clients. Such a schema can include containment hierarchies and the clients only have to be checked if a top-level class has to be changed.

Elimination of unnecessary relationships: In a containment hierarchy, there is already a strong relationship between a container and its children. A further dependency between the children is often unnecessary because the relationship from the parent can be used in most cases. The unnecessary relationship increases complexity. If the heuristic is violated, the relationship between the two classes is maintained in two different places: in the container and in the contained class. The potential redundancies results in an increased development effort.

Position in Life Cycle:

Design, Coding

Granularity:

Class in a containment hierarchy.

Example:

Violating:

```
class Car {
        int color;
        Wheels myWheels;
        Motor myMotor;
        void start(){
                myMotor.start();
        }
}
class Motor {
        Car myCar;
        void start(){
```

```
            while(!stop) myCar.myWheels.turn();
        }
}
class Wheels {
        void turn();
}
```

Complying 1:

```
class Car {
        int color;
        Wheels myWheels;
        Motor myMotor;
        void start(){
                myMotor.start();
                myWheels.turn();
        }
}
class Motor {
        Car myCar;
        void start();
}
class Wheels {
        void turn();
}
```

Complying 2:

```
class Car {
        int colour;
        Wheels myWheels;
        Motor myMotor;
        void start(){
                myMotor.start(myWheels);
                myWheels.turn();
        }
}
class TurnableItem{
        void turn();
}
class Motor {
        void start(turnableItem anItem){
                while(!stop) anItem.turn();
        }
}
class Wheels : turnableItem {
        void turn();
}
```


Subsuming/subsumed heuristic:

Not known.

Checking rules:

Find all contained classes. These classes have a relationship with the metarelationship *hasParts* or *hasAttributes*. In every contained class, find all relationships to other classes than the immediate child classes, e.g. *hasParts* or *hasAttributes* relationships to a base class or a sibling of the contained class.

```
select x,y, b.from,a
from x,y in type, a,b in relationship
where   a.myMetarelationship. name="sendsMessage"
and     b.myMetarelationship. name="hasParts"
and     a.from=x and y in a.to
and     x in b.to and y in b.to
```

Transformation rules:

Add a new relationship from the container to the contained class and remove the dependency between the contained classes.

```
insert b in b.from, y
remove b from x
```

Violated heuristics:

Not known.

Justifications for violating the heuristic:

Performance overhead: This heuristic results in more messages between objects. If messages are expensive in terms of performance (e.g. in a distributed environment where a message corresponds to a RPC (Remote Procedure Call)) this heuristic should be violated.

Reduction of uses relationships: Instead of multiple complex relationships between classes, a container can be introduced where classes depend. The container simulates the complex relationship.

Effects on measures:

Performance:	negative
Perceived complexity:	positive (few visible classes)
Isolateability:	positive
Dependencies:	positive

8.3 Every attribute should be hidden within its class

Do not define attributes in the public interface of a class.

Definitions:

Public interface: the public interface of a class is a list of operations and attributes that can be used from outside the class.

Rationale:

Object-oriented languages do not explicitly capture the relationship of a public attribute to the operations that use it. Therefore, it is difficult to determine which parts of the system depend on a public attribute. Every operation that can use the attribute has to be checked if the attribute changes. This makes the attribute rigid, especially if many different client classes/applications depend on it.

The owner of a class with a public attribute is not able to give guarantees about the value of the attribute if it is used by other client classes. The only way to ensure integrity constraints on the value is to control every location in every client who uses it, which is very ineffective. Public attributes in the schema of an objectbase avoid guarantees and make clients responsible to control and enforce integrity constraints, which is very ineffective.

Position in life cycle:

Design, Coding

Granularity:

Public interface of a class/module.

Example:

Violating:

```
class Asset{
public:
        double currentAmount, desiredAmount;
void order(int amount);
}
class AssetManager{
        Asset *myAsset;
        void adjustAsset(){
                if(myAsset->currentAmount < myAsset->desiredAmount)
                        myAsset.order(myAsset->desiredAmount- myAsset-
>currentAmount);
        }
}
```


Complying:

```
class Asset{
        private:
        double currentAmount, desiredAmount;
        public:
        void order(int amount);
        int needValue(){
                if (currentAmount < desiredAmount)
                return desiredAmount- currentAmount;
        }
}
class AssetManager {
        Asset *myAsset;
        void adjustAsset (){
                int amout=myAsset->needValue();
                if(amout>0) myAsset.order(amount);
        }
}
```

Subsuming/subsumed heuristics:

Subsumed: All data in a base class should be private(Riel 1996).

Checking rules:

Search public interface of every class/module for public attributes.

Transformation rules:

Replace the persistent attribute by two accessor operations (get/set). Move the persistent attribute in the private part of the class.

Violated heuristics:

8.7 Avoid pure accessor operations

Justifications for violating the heuristic:

Not known.

Effects on measures:

Size of class interface:	Negative
Rigidness of data structure:	Positive

8.4 Avoid dependencies of objectbase classes on their clients

Server classes should not depend on their clients. They should not use attributes or operations in the public interface of a client class in their opera-

tions. They should not inherit from a client class. They should not send messages to objects of a client class. Avoid dependencies of objectbase classes on classes outside the objectbase. Use observers or event mechanisms if the has to "talk" to its clients.

Definitions:

A client/server dependency is a dependency between a client class and a server class. The client class uses the server class. A server class is a class the objects of which receive messages from other objects. Objects of a client class send messages to objects of a server class.

Objectbase class: A class that can have persistent objects stored in an objectbase system. *Objectbase* classes are also called *persistence capable classes*. In most cases, they are elements of an objectbase schema. In other cases, objectbase classes are used in more general class schemas.

Event mechanism: An event mechanism is a technique that helps to decouple the creators of an event from the consumers of the event.

Rationale:

Reuse of abstraction: Dependencies of a server class on its clients raise the effort to use the server class in another context (e.g. from a different client class). This is because it is either necessary to eliminate these dependencies or to rebuild them in the new context. This is especially true for objectbase classes with many different client classes.

Cyclic instability: A server class should be stable. Dependencies make it rigid. If different kinds of client classes depend on a server class that violates this heuristic, a cycle of change propagation can occur. A change in the server class leads to a change in one of its clients which results in a subsequent change in the server (because it depends on the client) and so on.

Change propagation: In every client/server system with an objectbase, client classes use objectbase classes. We can distinguish two different cases:

- many classes from the same client use the objectbase classes and
- different clients use the objectbase classes.

Nearly every modification in the interface of an objectbase class leads to recompilation and redistribution of the depending clients. If an objectbase class depends on a client class, modifications in the client classes can result in subsequent modifications of the objectbase class. These changes often propagate to the interface of the objectbase class and lead to subsequent changes. Therefore, the effort for modifications and changes of the whole system becomes very high.

Database autonomy: One of the central requirements on an objectbase system, its autonomy, is violated.

Position in life cycle:

Design, Coding

Granularity:

Operations in server classes, Operations of objectbase classes.

Example:

Violating:

```
class DBcustomer{
private:
        Sting name;
        CustomerUI myUIs[];
        int index=0;
public:
        void subscribe(CustomerUI aUI){myUIs[index++]=aUI;};
        void setname(char* newname){
                strcpy(name, newname);
                for(i=0;i<index;i++) myUIs[i].nameChange();
        }
}
class CustomerUI{
        String customername;
        nameChange();
        printName(){print customername;}
        }
```

Complying:

```
class DBcustomer{
private:
        String name;
public:
        void set name(char newname){
                strcpy(name, newname);
                }
        String getName();
}
class customerUI{
        DBcustomer myC;
        printName(){print myC.getName();}
        }
```

Subsuming/subsumed heuristic:

Not known.

Checking rules:
Find all operations of a server class that send messages to client operations. Check operations of objectbase classes for use of client classes.

Transformation rules:
Observer Transformation: This transformation converts the server class into a concrete subject and the client class into a concrete observer in the observer pattern.

Assume that the client class is called client and the server class is called server. Define a new class serversubject Define an inheritance relationship from server to serversubject Define a new class clientobserver Define an inheritance relationship from server to clientobserver Define a 0-n aggregation from serversubject to clientobserver Add an operation GetState() to server Add an operation Update to client. Update uses GetState()

Violated heuristics:
8.7. Avoid pure accessor operations

Justifications for violating the heuristic:
Not known.

Effects on measures:

Coupling between classes:	Positive
Coupling of objectbase classes to client classes:	Positive

8.5 A class should capture one, and only one key abstraction with all its information and its entire behavior

Do not distribute knowledge about an important abstraction among a lot of classes. Do not model different important abstractions in one single class.

Definitions:
Key abstraction: A key abstraction represents an entity within the perceived mini world.

Rationale:
If two key abstractions A, B are modeled within one class C, this often leads to redundant attribute *values* within C. This is because an object *i*(A) of A

can be combined with different objects of B and vice versa. Every combination containing *i*(A) includes *i*(A)'s information.
This kind of redundancy is not desired. Violation of data consistency in objects of C is possible. All updates to *i*(A)'s information have to be propagated to all combinations of *i*(A).

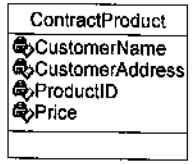


Figure 36: ContractProduct

Consider the class *ContractProduct* in Figure 36. In this class, three key abstractions (*Contract*, *Product*, *Customer*) are modeled. The attribute value of *CustomerAddress* for a given customer will occur redundantly if more than one object contains the same customer. Therefore, the data consistency can easily be violated .

If a single key abstraction is distributed among a set of classes it is harder to find locations where changes have to be applied. Often some of the classes are very small and can be eliminated.

Position in Life Cycle:
Analysis, Design, Coding

Granularity:
Class

Example:
Violating:

```
class ProviderProduct{
        string providername, provideraddress;
        string productname, price;
}
```

Complying:

```
class Provider{
        string providername, provideraddress;
        set of Product offeredProducts;
}
class product{
        string productname, price;
}
```

Subsuming/subsumed heuristic:
Not known.

Checking rules:
Check if data members are distributed in proper subsets by their using operations.
Algorithms to check for 3nf based on functional dependencies.
Remember that most of the operations should use most of the data members most of the time.

Transformation rules:
Decomposition:
1. Decompose a single class based on subsets of attributes used by its operations.
2. Use decomposition algorithms and FDs (Functional Dependencies) (see 8.9)
3. Decompose a single class based on subsets of attributes used by operations of other classes

Composition: Create an URW-matrix. A URW (Using, Read, Write) matrix consists of a row for every operation and every attribute and a column for every method. The row of an attribute contains an R if the attribute is read by the other method, a W if the attribute is written. The row of an operation contains a U if this operation is used by the other method. Find patterns in the columns of the matrix. Such a pattern indicates that different attributes of operations are always used in the same way. If such attributes or operations are located in different classes, these patterns describe possible class compositions.

Violated heuristics:
Not known.

Justifications for violating the heuristic:
Performance considerations: Many dereferenciations

Effects on measures:
Number of classes: Negative
Average size of class: Positive

8.6 Do not create unnecessary classes to model roles

Entities with different roles in the miniworld are often described in different classes. In this case, the classes mirror a subset of the roles of an entity. These different classes often have the same properties and operations. From

a design perspective, they are not different, e.g. an address register only contains addresses of persons; teenagers and twens do not differ. If there are *no or nearly no new attributes and operations* in the subclasses, one class is sufficient to describe all roles and classes. Mirroring the roles of the entity in additional classes is unnecessary and leads to additional class overhead. Model only one class for an entity with different roles, and provide the role information in other ways, e.g. in state attributes.

Definitions:
Role: A role is a defined view on an entity. Its class describes the role an entity plays. Objects represent the different roles of the entity.
Entity plays different roles: An entity always plays a role. In some cases, entities in the miniworld play different roles. Trainer and employee are typical roles for a person.

Rationale:
Modeling different roles of an entity in unnecessary classes leads to derived classes that share common attributes and behavior. They do not provide interesting properties and unnecessarily extend the size of the model.
Selection Anomalies: In some environments, role classes lead to selection anomalies; this means different selections on different classes have to be executed and merged to collect all objects of a class.
Type migration: A role class defines a *static type* for its objects. In most objectbases, type-migration (e.g. a tennager becomes a twen) is a complex task. The existing object of the old type has to be deleted[16], and a new object has to be created. If an attribute is used to define the type, the type migration is a value change of the attribute.

Position in Life Cycle:
Analysis, Design, Coding

Granularity:
Classes in a schema.

Example:
Violating:

```
class Dog{
        string name, colour;
        void wag_tail();
}
class Homedog: Dog{}
class Workdog: Dog{}
```

Complying:

[16] Lack of techniques to define and enforce referential integrity constraints in most objectbase systems leads to additonal trouble.

```
class Dog{
        string name, colour;
        void wag_tail();
        int dogtype;
}
```

Subsuming/subsumed heuristic:
Not known.

Checking rules:
Check all base classes for derived classes with the same properties.

Transformation rules:
1. Delete the derived classes and put additional properties in base class.
2. Transform the different roles into views on a common entity.

Violated heuristics:
8.19 Avoid case analysis on properties of objects, 8.25 Avoid direct recursive associations

Justifications for violating the heuristic:
A performance trade-off makes it necessary to separate a large set of objects into different subsets. If the role classes have additional interesting properties and operations, the heuristic should be violated.

Effects on measures:
Size of schema: Positive

8.7 Avoid pure accessor operations

Try to minimize the number of operations that do nothing else than returning or changing an attribute value. Instead of pure accessor operations, use operations which implement some interesting behavior of the object.

Definitions:
Accessor method: A public operation that reads or writes an attribute of its class.
Pure accessor method: A public operation that only reads an attribute of its class or only modifies an attribute of its class.

Rationale:
Redundant code: A pure accessor operation exports control of an attribute to other classes. This allows implementers of these other classes to mix behavior of different classes in a single method. Furthermore, pure accessor

operations encourage code redundancies. That is, the same functionality (which would belong to the class of the pure accessor) can be implemented in different locations. (Riel 1996) notes that pure accessors indicate poor encapsulation of data and behavior.

Information hiding: Pure accessors do not provide information hiding. Clients of a class with pure accessors have complete control over the accessed attribute values. The owner of the class cannot guarantee that the information provided by the attribute remains correct.

Position in Life Cycle:
Analysis, Design, Coding

Granularity:
Operations of classes

Example:
Violating:

```
class Credit{
private:
        float value;
        float securityValue;
        float evaluation;
public:
        float getValue() {return value;}
        float getSecurityValue() { return securityValue;}
        void  setEvaluation(int eval);
}
class CreditChecker{
        creditlist myCredits[];
        int evaluateCredits(){
                for all elements c in myCredits{
if(c.getValue()-c.getSecurityValue()>100000) c.setEvaluation(BLACK);
                        else            if(c.getValue()-c.getSecurityValue()<0)
c.setEvaluation(WHITE);
                        else c.setEvaluation(GREY);
                }
        }
}
```

Complying:

```
class credit{
private:
        float value;
        float securityValue;
        float evaluation;
public:
```

```
        void evaluate(){
                if(value-securityValue>100000) evaluation=BLACK;
                else if(value-securityValue<0) evaluation=WHITE;
                else evaluation=GREY;
        }
}
class CreditChecker{
        creditlist myCredits[];
        int evaluateCredits(){
                for all elements c in myCredits c.evaluate();
        }
}
```

Subsuming/subsumed heuristic:
Not known

Checking rules:
Check each class for operations that have the same name as an attribute of the class that is extended by the prefix *get* or the prefix *set*.

Transformation rules:
If an accessor operation is found, find all locations in which it is used. Check if further transformations are possible.
The following questions should be answered: Why do other classes work on the accessor operation class? Can the class itself also do the work?

Violated heuristics:
Not known

Justifications for violating the heuristic:
A class directly used by a user interface that has to show and enter data.

Effects on measures:
Size: Positive

8.8 Avoid additional relationships of base classes to their derived classes

Base and derived classes have already a relationship that implies a strong dependency of the derived to the base class. Avoid associations and using relationships that lead to a dependency from a base class to its derived class.

Definitions:

Relationship: Classes have different kinds of relationships to each other. Examples of relationships are inheritance, associations and composition. Each relationship leads to a dependency. E.g. the inheritance relationship leads to a dependency of the derived class to the base class, i.e. if an operation interface in the base class has to be changed, the derived class has to be changed too.

Rationale:
The derived class inherits the additional relationship, so besides the relationship between super and derived class, there is a recursive relationship from the derived class to itself. Even if the relationship (e.g. association) is private, there are often public operations, e.g. *Product.getMyProductOffers()* that use it. These operations are also inherited, which in many cases makes no sense.
Furthermore, if a base class sends messages to a derived class, a high level concept (expressed by the base class) depends on a low-level concept (expressed by the derived class). This leads to further changes that are not desired if new classes are added.

Position in Life Cycle:
Analysis, Design, Coding

Granularity:
classes and their relationships

Example:
Violating:

```
class Product {
ProductOffers myProductOffers;
}
class ProductOffer : Product {
}
```

Complying:

```
class Product : Catalogitem {
ProductOffers myProductOffers;
}
class ProductOffer : Catalogitem {
}
```

Subsuming/subsumed heuristic:
Subsumed by: Avoid concrete base classes (Riel 1996).

Checking rules:
Check all classes with derived classes for relationships to their derived classes.

Transformation rules:
Remove the inheritance relationship. Create a new abstract base class with the properties of the former base class for both of the classes.

Violated heuristics:
Not known

Justifications for violating the heuristic:
Not known

Effects on measures:
Number of classes: Negative
Coupling: Positive

8.9 Avoid classes with properties implying redundancies

Description:
This heuristic is a transformation of the ideas of 3nf and BCNF (Bernstein 1976; Fagin 1977; Elmasri, Navathe 1989) to the context of objectbase design. Let c be a class. Let X->A be a FD in c, where X is a subset of the attributes of c, and A is an attribute of c. Either X should be an identifier of c, or A should be a prime attribute.

Definitions:
Functional dependency (FD): Let *A={A1,..., An}* be the set of properties of a class c. A functional dependency denoted by *X->Y*, between two sets of properties *X={X1, ..., Xk}, Y={Y1, ...,Ym}* with *X, Y* in *A* is a constraint on the possible values of the properties of *c*. It exists between *X* and *Y* if each set *x1,...,xk* of values of *X1,...,Xk* corresponds to precisely one set of values *y1,...,ym* of the attributes *Y1,...,Ym*.

Identifier of a class c: Let *o* be an object of *c* and *o[X]* be the tuple of values of *X* in *A*. Let *I* be a subset of *A*. *I* is called identifier of *c* if *o1[I]!=o2[I]* for any two distinct objects *o1, o2* of *c*.

Prime attribute: A key *K* of *c* is an identifier with the additional property that removing any attribute from *K* leaves a set of attributes which is not an identifier. An element of a key is called prime attribute.

Transitive FD: A FD *X->Z* is transitive if there exist *X->Y* and *Y->Z*, so that *Y* is not a subset of a key.

Rationale:
A violation of this heuristic means that a FD *X->A* exists; *X* is not an identifier of *c* and *A* is not a prime attribute. *X* could be nonprime or a proper subset of a key. If *X* is nonprime, we have a *transitive dependency*, if *X* is a proper subset, we have a *partial dependency*.

If we have a *partial dependency* it is possible to identify a subset of the attribute values of every object by a part of a key. This can easily lead to data redundancies. This is because the same objects (representing the same information) of the two concepts are bundled in different ways. This redundancy can lead to update anomalies. If an attribute value changes, all objects have to be changed (*"modification anomaly"*), if the last object representing one of the two concepts is deleted, the information about the other one is also lost (*"deletion anomaly"*). Furthermore, this suggests that we have modeled two different concepts in one class. This is because two different identifiers for different attribute sets exist.
If we have a transitive dependency, a subset Z of the attribute values can be identified by a non key *Y* and the non key can be identified by another attribute set *X*. This situation also leads to data redundancies and it indicates that two classes, one containing the attributes *XY* the other the attributes *YZ*, are modeled in one class.

Position in Life Cycle:
Analysis, Design, Coding

Granularity:
Attributes of classes.

Example:
Violating:

```
class CustomerContract{
        string customerName;
        string customerNumber;
        string contractNumber;
        string contractValue;
}

Primary key: customerNumber, contractNumber
Prime attributes: customerNumber
FDs: customerNumber->customerName, contractNumber->contractValue
```

Complying:

```
class Customer{
```

```
        string customerName;
        string customerNumber;
}

class Contract{
        string contractNumber;
        string contractValue;
        string customerNumber;
}
```

Subsuming/subsumed heuristic:
Not known

Checking rules:
Implied by the definitions above: check every FD for the required properties.

Transformation rules:
If possible, transform the depending attributes to operations that embody the functional dependency. In other cases, split up class by 3nf-normalization algorithm and rules from 3nf (Elmasri, Navathe 1989).

Violated heuristics:
Not known.

Justifications for violating the heuristic:
Not known.

Effects on measures:
Not known.

8.10 Avoid multivalued dependencies

Eliminate every multivalued dependency.

Definitions:
A *multivalued dependency (MD)* *X=A->>B* between two sets *A=(A1,...,An)*, *B=(B1,...,Bm)* of all attributes *Z* of a class *c* exists if *A* determines a set of values for the attributes *B* of objects of *c* completely. Let *i* an object of *c* with i[A]=(a1,...an) and i[B]=(b1,...bn). The MD *X=A->>B* specifies the following constraint on the set of objects of *c*: For every object j of c the statement j[A]=i[A] => j[B]=i[B] is valid.

Rationale:

The set of values of *B* is completely determined by the value of *A*. This means that the value of *A* is repeated for every value of *B*. This is a redundancy which can cause update anomalies, and objects of these classes always have to be treated as a group, e.g. for creation, deletion and other operations. The classes which violate this heuristic have to be grouped to form a single entity.

Position in Life Cycle:
Analysis, Design, Coding

Granularity:
What kind of item (module, class and method) is effected? What do I have to check to find a violation?

Example:
Assume that a single contract includes a single customer and many products.
Violating:

```
class Contract{
        int customerNumber;
        int productId;
}
```

Complying:

```
class Contract{
        int customerNumber;
        set(int) productsIds;
}
```

Note: The information in the class specification is not sufficient to detect multivalued dependencies. This makes it impossible to violate 4NF.

Subsuming/subsumed heuristic:
Not known

Checking rules:
Not known. The designer has to specify multivalued dependencies.

Transformation rules:
Aggregation transformation: Create a new set attribute that contains *structs* of *B* attributes.
Create a new class *C* containing all *B* attributes. Create a set attribute that contains objects of *C*.

Violated heuristics:

Not known

Justifications for violating the heuristic:
Not known

Effects on measures:
Not known

8.11 Whenever possible, convert associations and uses relationships in the strongest containment relationship

Avoid *loose* relationships. Replace loose relationships by relationships that restrict visibility. Question symmetric relationships: expect asymmetry.

Definitions:
Visibility: Visibility is a term that describes which parts of a fragment are usable by another part of a fragment. E.g., in an operation, only the current objects (this, self, etc, ...), objects defined by relationships and objects defined in the parameterlist are visible.

Strong/loose relationship: We define the strengths of a relationship as the amount of restrictions (e.g., a containment relationship restricts the visibility of the contained objects to the container objects) the relationship carries. A loose relationship carries few restrictions. Examples for restrictions are the visibility and the cardinality of the relationship. A relationship with cardinality 1 is stronger than a relationship with cardinality (0,n).

Rationale:
Complexity can be divided in *essential complexity* and *unnecessary complexity*. *Essential complexity* origins in the application domain and cannot be reduced, only managed. *Unnecessary complexity* origins in wrong use of techniques; e.g. modeling constructs or tools. A design with much unnecessary complexity is a bad design. Unnecessary complexity has to be reduced in a good design. A possible source of unnecessary complexity is a loose relationship that can be replaced by a stronger relationship that reduces visibility or specifies more detail. Examples of loose relationships are reference associations, especially symmetric associations.

Furthermore, strong dependencies allow exploiting the support for complex objects in many objectbase management systems. These systems allow optimizing storage of embedded objects (e.g., they store contained objects on the same page as the container objects).

Position in Life Cycle:
Analysis, Design, Coding

Granularity:
Classes and relationships

Example:
Violating:

```
class Car{
public:
        set(Engine) engines;
}
class Engine{
}
```

Complying[17]:

```
class Car{
private:
        Engine myEngine;
}
class Engine{
}
```

Subsuming/subsumed heuristic:
Not known

Checking rules:
Association detail:
Find all n to n associations. Ask designer to replace by asymmetric associations.

Visibility reduction:
For each class, find all classes it is associated to. In the associated classes, find groups of classes which are not used by other classes outside the group.

Transformation rules:
Visibility reduction:
Transform the uses relationships in this group to a containment relationship. Check real world with designer; if violated, try to find a new containing class.

[17] Note: in this example, the complying fragment is no equivalent to the violating fragment. Only the transformation from a weak to a stronger association is illustrated (based on the assuption that a car has only one engine which is not visible outside the car).

Violated heuristics:
Not known.

Justifications for violating the heuristic:
This heuristic can be violated if the same object has to be shared by many other objects.

Effects on measures:

Perceived complexity:	Positive
Strength of dependency from other classes:	Positive
Efficiency:	Negative
Isolateability:	Positive

8.12 Avoid contained objects that can concurrently be modified

To modify the state of an object, objectbase transactions lock the object. Avoid a class specification where different contained objects have to be modified by concurrent transactions.

Definitions:
Contained object: An object is called contained object if a container object exists. A container object manages its contained objects. This means it has references to the contained objects, and many operations use the contained objects. In a strong containment, the container is the only object which can use its contained objects.

Complex attribute value: A complex attribute value is an attribute value which consists of more than one simple value (e.g. int 1). Examples for complex attribute values are a set of Customers or a structure consisting of two integer values.

Object valued attribute: An object values attribute is an attribute whose value is an object, e.g. a customer or a contract.

Rationale:
Assume two concurrent, different transactions t1, t2 where t1 modifies the contained object a1 and t2 the contained object a2 of an object o. During the time in that t1 is executed o is locked. Therefore, t2 has to wait to modify a2. If a1 and a2 where not contained in o, t1 would only lock a1, and t2 could concurrently modify a2.

Position in Life Cycle:
Analysis, Design, Coding

Granularity:
Attributes of a class

Example:
Violating:

```
class Stock{
private:
        set(struct<date, time, value>) courseHistory;
        CourseTimeSeries  myEstimation;
public:
        void enterCourse(date, time, value);
        void enterEstimation(date, time, value);
}
class CourseTimeSeries{
        set(struct<date, time, value>) series;
}
```

Complying:

```
class Stock{
private:
        CourseHistory * courseTimeSeries;
        CourseEstimation  *myEstimation;
public:
        enterCourse(date, time, value)
        enterEstimation(date, time, value)
}
class CourseTimeSeries{
        set(struct<date, time, value>) series;
}
```

Subsuming/subsumed heuristic:
Not known

Checking rules:
Assume that operations of objectbase classes contain transactions. Identify complex attributes modified by objectbase operations. Find overlapping parts in these attributes. Ask designer if attributes are concurrently modified.

Transformation rules:
Split the complex attributes into simpler ones, or in references to objects which contain fewer overlapping parts.

Violated heuristics:
Not known.

Justifications for violating the heuristic:
Not known.

Effects on measures:
Schema performance: Positive

8.13 All properties of the basetype must be usable in objects of its subtypes in every location in that a basetype object is expected

(The Liskov Substitution Principle (Liskov 1987))
The subtype (Abadi, Cardelli 1996) should fully implement all of its basetypes. If a basetype object is expected, no additional properties of a subtype should be needed. Avoid an inheritance relationship if some of the inherited properties are not valid for the objects of the subtype. Not valid means that for an object of the subtype, the operation has a semantic that requires some kind of special treatment in client operations. Avoid a subtype relationship if the subtype is a specialization of the basetype that cannot support the complete basetype interface.

Rationale:
If this heuristic is violated, special treatment is required if an object of a subtype is used in an operation where an object of a basetype is expected. This means, that the implementation of a (high level) operation expecting a basetype object depends on a (lower level) subtype object. This is generally bad design. Furthermore, this means, that adding new subtypes to the basetype leads to subsequent changes in the operation that requires special treatment. This makes the basetype hard to extend. This is also a violation of the open/closed principle (Meyer 1988). The basetype is not open to extensions (by adding a new subtype) and closed for modifications because some operation has to be changed. Adding a new subtype is an extension that should always be possible without the need for subsequent modifications.

Position in Life Cycle:
Analysis, Design, Coding

Granularity:
Two types with an inheritance relationship.

Example:

The classical example is the emu example. An emu is a bird, but it cannot fly.

Violating:

```
class Bird{
        int fly(){ behavior(); return flying } ;
}
class Emu{
        int fly(){ return cannot_fly; }
}
```

Complying:

```
class Bird{}
class FlyingBird:Bird{
        fly();
}
class NonflyingBird:Bird{}
class Emu: NonFlyingBird{}
```

Another example is:
Violating:

```
class Asset{
        float interest_rate;
}
class VisaAccout: Asset{
}
```

Complying:

```
class Asset{
}
class InterestAsset:Asset{
        float interest_rate;
}
class NonInterestAsset:Asset{}
class VisaAccount: NonInterestAsset{}
```

Subsuming/subsumed heuristic:
Not known

Checking rules:
Check if there is an operation that depends on both the basetype and the subtype. This operation can be an operation of the bastype, the subtype or some other type. Check if the dependency from the subtype origins in the need to modify something.

Transformation rules:

Replace the basetype by two new types. Move the property to one of the new types and assign the correct inheritance relationships.

Violated heuristics:
Not known

Justifications for violating the heuristic:
Explosion of inheritance hierarchy: This transformation should be executed if new reasonable types are discovered. If this transformation is often executed (for single operations), the inheritance hierarchy might explode and contain a lot of classes which are only separated by simple operations. This situation should not occur.

Effects on measures:
Coupling: Positive
Size: Negative

8.14 Common properties of objects should be defined in a single location

Description:
Avoid properties that have the same meaning but are defined in different locations. Move common properties in derived classes to the base class.

Definitions:
Common properties: Two properties are common when they have the same name and the same meaning.

Rationale:
If properties are common, defining them in different locations results in non-desirable redundancies. These redundancies lead to unnecessary effort if the properties have to be changed, because the properties to be changed have to be found in many locations and the change has to be executed in many locations instead of a single one.

Position in Life Cycle:
Analysis, Design, Coding

Granularity:
Properties of classes

Example:

Violating:

```
class Student{
        string name;
}
class Employee{
        string name;
}
```

Complying:

```
class Person{
        string name
}
class Student : Person {}

class Employee: Person {}
```

Violating:

```
class Boat{
        int x_position;
        int y_position;
}
class Airplane{
        int x_position;
        int y_position;
}
```

Complying:

```
class Position{
        int x;
        int y;
}
class Boat{
        Position myPosition;
}
class Airplane{
        Position myPosition;
}
```

Complying:

```
class Vehicle{
        int x;
        int y;
}
class Boat : Vehicle{}
class Airplane : Vehicle{}
```

Note:
Sometimes, the structure of the common properties is different, which is claimed to be the reason that the common properties are in different locations. Consider this example from a recent discussion:

```
class Tuna{
        void swim(int x, int y);
        void breathe( water w);
}
class Cow{
        void walk( int x, int y);
        void breathe( air a);
}
```

The author claims that a solution is an empty animal base class. The class is empty because there are no common properties. We argue that if we define a class animal (which is an abstraction of cow and tuna), we also have to put the properties (and the structure of the properties) on a higher level of abstraction, e.g.

```
class BreatheableMatter {}
class Air: BreatheableMatter {}
class Water: BreatheableMatter {}
class Animal{
        void move(int x, int y);
        void breathe(BreatheableMatter b);
}
class Cow: Animal{
}
class Tuna: Animal{
}
```

Subsuming/subsumed heuristic:
Not known

Checking rules:
Check classes for similar names/structures of attributes.

Transformation rules:
Create a new class that includes these similar properties.
Define it as a base class for the classes with the similar properties if no inheritance heuristics are violated. Otherwise, define associations to the new class.

Violated heuristics:
Not known

Justifications for violating the heuristic:
Not known

Effects on measures:
Not known

8.15 Soft classes should not be base classes

Do not use a coupling-soft class as a base class of other classes. These classes are changed more often than coupling-hard classes. A class with a low value of HF (see section 7.9) should not be a base class.

Rationale:
The main sources of coupling-softness are *dependencies* from other classes. If one of the other classes is changed, the change may have to be propagated to the depending class. The main concerns if a soft class is a base class is that changes in the class are propagated to its derived classes (and lead to subsequent changes in their clients).

Position in Life Cycle:
Analysis, Design, Coding

Granularity:
Single classes

Example:
Violating:

```
class Credit {}
class SmallCredit: Credit {}
class CreditManager{
        set(Credit *) myCredits;
        createCredit(){
                new credit();
        }
}
```

Complying:

```
abstract class Credit {}
class OrdinaryCredit: Credit {}
class SmallCredit: Credit {}
class CreditManager {}
```

```
        set(Credit *) myCredits;
        createCredit(){
                new OrdinaryCredit();
        }
}
```

Subsuming/subsumed heuristic:
Not known

Checking rules:
Calculate HF for all classes in the schema. Define an upper limit. Check if classes above the limit are base classes.

Transformation rules:
1. Define new base class.
2. Make the former base class a leave class.
3. Either reduce the coupling for the class or make it a leave class.

Violated heuristics:
Not known

Justifications for violating the heuristic:
Not known

Effects on measures:
Size: Negative
Coupling: Positive

8.16 Do not misuse inheritance for sharing attributes

Description:
Avoid using inheritance if you intend to share only the attributes of the inherited base class among the derived classes. Use association or aggregation of a shared object for sharing attributes.

Rationale:
If attributes are shared in two classes by inheritance, the value of the attributes is *located at two places*. Consider the classes *Credit, Private credit* and *Company credit* where *Credit* inherits from *Security*. *Private* and *Company credit* inherit the security data. Questions like "who used this house as a security" are hard to answer. Instead of this, use a contained object *Security* which only exists once (at a time).

Position in Life Cycle:
Analysis, Design

Granularity:
Classes with inheritance

Example:
Violating:

```
class Position{
        int x, y;
}
class Car : Position{ }
```

Complying:

```
class Position{}
class Car{
        Position myPosition;
}
```

Subsuming/subsumed heuristic:
Not known

Checking rules:
Check if there are base classes that have only attributes.

Transformation rules:
Define a new class with the shared attributes. Use this class as a content contained in both classes.

Violated heuristics:
Not known

Justifications for violating the heuristic:
Not known

Effects on measures:
Not known

8.17 The overloading should only define differences to the overloaded operation

Exactly analyze the differences in both operations, then describe these differences in the overloading operation.

Definitions:

Overloaded operation: An operation for which an operation with the same signature is defined in a derived class.
Overloading operation: The corresponding operation in the derived class.

Rationale:

- Reduces the amount of duplicate behavior. Duplicate behavior has to be changed in duplicate places.
- Allows for behavior extension of derived classes by adding new operations in the base class

Position in Life Cycle:

Design

Granularity:

Operations of classes

Example:

Smalltalk classes print function and string representation.
Violating:

```
class Person{
        string name;
        string address;
        void print (){print name+address}
class Student: Person {
        string studentnumber;
        void print(){ super.print(); print studentnumber; }
```

Complying:

```
class Person{
        String name;
        String address;
        String toString() { return name + address; }
        void print() { print(toString()); }
class Student
        String studentnumber;
        String toString() {return super.externalize() + matnr }
```


Subsuming/subsumed heuristic:

Not known

Checking rules:

Find at least two overloaded operations that use the same attributes.
Check if the attributes are used for the same computation.

Transformation rules:

Create a new operation in the base class that includes the common computation.

Violated heuristics:

Not known

Justifications for violating the heuristic:

Not known

Effects on measures:

Not known

8.18 Avoid full parallel overloading in siblings

Avoid overloading the same operation(s) in all derived classes of a base class.

Definitions:

Siblings: Siblings are derived classes of the same class which have the depth in the inheritance tree.

Rationale:

Often, the same operation is overloaded in many derived classes of the same class. This is usually a hint for the fact that an abstraction is missing in the design of the derived classes. Such a missing abstraction results in a situation in which the code of the overloading operations has a very similar structure. Often, the structure of the code is the same, and the only differences are the types of the used objects. This structural redundancy is dangerous and not desirable. If the structure of the code has to be changed, it has to be changed in all overloading operations.

Position in Life Cycle:

Design

Granularity:
Operations of classes

Example:
Violating:

```
class Manager{
        print (){}
}
class StudentManager: Manager
        set(Student) myStudents;
        int getAvgNote();
        print(){ myStudents.print(); }
}
class EmployeeManager: Manager {
        set(Employee) myEmployees;
        int getAvgIncome();
        print(){ myEmployees.print(); }
}
```

Complying:

```
class Manager{
        set(Object) myManagedObjects;
        print (){myManagedObjects.print() }
}
class StudentManager: Manager{
        int getAvgNote();
}
class EmployeeManager: Manager{
        int getAvgIncome();
}
```

Subsuming/subsumed heuristic:
Not known

Checking rules:
1. For all derived classes of a base class: find all overloaded operations with the same name and the *same* parameterlist.
2. For all derived classes of a base class: find all overloaded operations with the same name and a *different* parameterlist.

Transformation rules:
Define a class for the missing abstraction. Add a reference to this class for the base class. Abstract the redundant implementation and remove it from the derived classes.

Violated heuristics:

Not known

Justifications for violating the heuristic:
This heuristic may be violated if the size and the effort to create the new abstraction is larger than the overhead from the redundant overloading operations. This is typically the case if only very few operations are redundantly overloaded.

Effects on measures:
Not known

8.19 Avoid case analysis on properties of objects

Avoid case analysis on attributes of an object which influence its behavior.

Rationale:
Case analysis on properties of an object often results in major drawbacks:
1. it is hard to add new sibling classes to classes of the object
2. changed attribute values require modifications in the case analysis

Position in Life Cycle:
Design

Granularity:
Attributes and operations of classes

Example:
Violating:

```
class Credit{
        int value;
        int securityValue;
        CustomerEvaluation c;
        int getCustomerEvaluation();
        int evaluateCredit(){
                if(value-securityValue>100000) return BLACK
                else if(value-securityValue<0) return WHITE
                else return GREY;
        }
}
class CreditChecker{
        void check(){
                for all credits c{
```

```
                    switch(c.evaluateCredit()){
                    case BLACK: infomOfficer(); break;
                    case GREY:
                        if(c.getCustomerEvaluation()==NEGATIVE)
THEN
                            informAssistant();
                            else informCustomer();
                    break;
                    }
        }
}
```

Complying:

```
class CreditFactory(){
        void createCredit(int value, int securityValue){
            if(value-securityValue>100000) new blackCredit();
            else if(value-securityValue<0) new whiteCredit();
            else new greyCredit();
        }
class Credit{
        int value;
        int securityValue;
        check(){ return;}
}
class BlackCredit{
        check(){infomOfficer();}
}
class   WhiteCredit();{}
class GreyCredit(){
        check(){
            if(getCustomerEvaluation()==NEGATIVE) THEN
            informAssistant();
            else informCustomer();
        }
}
class CreditChecker{
        Check(){
            for all Credits c
                c.check();
}
}
```

Subsuming/subsumed heuristic:
Not known

Checking rules:
Check for case statements that switch over attributes or operations of an object.

Transformation rules:
Propose new classes based on the case clause.

Violated heuristics:
8.20 Prefer typing by attribute to typing by inheritance, 8.6 Do not create unnecessary classes to model roles

Justifications for violating the heuristic:
Avoid classes without functionality: see rationale of 8.20 and 8.6.

Effects on measures:
Not known

8.20 Prefer typing by attribute to typing by inheritance

Similar objects should have the same class. Objects are similar if they have similar properties (operations and attributes). Often, there is a large set of objects with identical properties and a small set of objects with many differences. If the differences are small the differing objects should be modeled in the same class. Attributes should be used to model the differences.

Definitions:
Typing by attribute: An attribute is used to define a type of an object, e.g., the attribute *customerkind* in the class *Customer*.
Typing by inheritance: Inheritance is used to define the type of an object, e.g., the classes *PriveteCustomer* and *RetailCustomer* are defined as derived classes of the class *Customer*.

Rationale:
Unnecessary classes have many drawbacks:

- class migration and dangling references if object changes its class
- complicated creation of an object

This heuristic is only the more important if an object-oriented database management system is used. Class migration of objects is always a serious issue, because all client objects maintaining references to an object have to be updated if the object changes its class. This heuristic proposes to separate

the object from differing behavior. The object stays the same. The behavior is different. The object references stay the same.

Position in Life Cycle:
Design

Granularity:
Attributes of a class.

Example:
Violating:

```
class CreditFactory{
        Credit createCredit(int theValue, int theSecurityValue){
                if(theValue-theSecurityValue>100000)
                        return new BlackCredit(theValue, theSecurityValue);
                else if(value-securityValue<0)
                        return new WhiteCredit(theValue, theSecurityValue);
                else return new GreyCredit(theValue, theSecurityValue);
        }
        void checkAll(){
                for all Credits c
                        c.check();
        }
}
class Credit{
        int value;
        int securityValue;
        new (int theValue, int theSecurityValue){
                value=theValue;
                securityValue=theSecurityValue;
        }
        check(){ return;}
}
class BlackCredit: Credit{
        check(){informOfficer();}
}
class WhiteCredit: Credit{}
class GreyCredit: Credit{
        check(){
                if(getCustomerEvaluation()==NEGATIVE)
                informAssistant();
                else informCustomer();
        }
}
```

Complying:

```
class CreditFactory{
        Credit createCredit(int theValue, int theSecurityValue){
                return new Credit(theValue, theSecurityValue)
        }
        void checkAll(){
                for all Credits c
                        c.check();
        }
}
class Credit{
        int value;
        int securityValue;
                check(){
                new Handler(value-securityValue).check();
        }
        new (int theValue, int theSecurityValue){
                value=theValue;
                securityValue=theSecurityValue;
        }
class Handler{
        int valueWithOutSecurity
        new (int theValueWithOutSecurity){
                valueWithOutSecurity=theValueWithOutSecurity;
        }
        check(){
                if(valueWithOutSecurity>100000)
                        infomOfficer();
                else if(valueWithOutSecurity<0)
                        return;
                else {
                        if(getCustomerEvaluation()==NEGATIVE)
                                informAssistant();
                        else informCustomer();
                }
}
```

Subsuming/subsumed heuristic:
Not known

Checking rules:
Not known.

Transformation rules:
Not known.

Violated heuristics:
8.19 Avoid case analysis on properties of objects

Justifications for violating the heuristic:
Exception classes in Java (Gosling, Joy et al. 1996; Gosling, Joy et al. 2000).

Effects on measures:
Not known

8.21 An operation should only use classes of attributes of its class, classes of its parameters or classes of locally created objects

This is an application of the law of Demeter (Lieberherr 1992). The implementers of an operation should only use this restricted set of classes in the operation implementation.

Definitions:

Rationale:
Following this heuristic has two main reasons.
If the heuristic is violated, *dependencies between classes that exist in a class hierarchy are encoded redundantly* in an operation by so-called path expressions. E.g., the path expression *myCustomer->getMainPortfolio()->getAssetlist()->sellAssets();* found in an operation of the *CustomerManager* class encodes the information that a *Customer* class has an association with a *Portfolio* class and that the *Portfolio* class has a list of objects of the *Asset* class and that this list has an operation *sellAssets*(). This information is redundantly contained in the class hierarchy. If we need to change, e.g., the name of *theAssetlist* to *anAssetlist*, we have to change, recompile and test the *AccountManager* class. This makes it very hard to change the class structure, because the redundant relationships have to be changed, too.
The possible dependencies between an operation and other classes are reduced. This means, only a limited amount of operations has to be checked if a class is changed, and the amount of possible change propagations is reduced.

Position in Life Cycle:
Analysis, Design Coding

Granularity:
Operations of classes

Example:

Violating:

```
class CustomerManager{
        Customer myCustomer;
        void makeLiquid(){
        myCustomer->getMainPortfolio()->getAssetlist()->sellAssets();
        }
}
class Customer{
private:
        Portfolio mainPortfolio;
public:
        Portfolio getMainPortfolio()
}
class Portfolio{
private:
        Assetlist theAssetlist;
public:
        AssetList getAssetList();
}
class Assetlist{
        set of Asset myAssets;
        void sellAssets();
}
class Asset{}
```

Complying:

```
class CustomerManager{
        Customer myCustomer;
        void makeLiquid(){
                myCustomer->makeLiquid();
}
}
class Customer{
        Portfolio mainPortfolio;
        void makeLiquid(){
                mainPortfolio->sell();
}
class Portfolio{
        Assetlist theAssetlist;
        void sell(){
                theAssetlist->sellAssets();
}
}
class Assetlist{
        set(Asset) myAssets;
        void sellAssets();
}
```

```
class Asset{}
```

Subsuming/subsumed heuristic:
Not known

Checking rules:
Use checking mechanisms from Demeter tools (Lieberherr 1992). Check path expressions in operation collaborations.

Transformation rules:
Replace path expressions by creating new operations and using these operations in simple paths.

Violated heuristics:
Not known

Justifications for violating the heuristic:
Not known

Effects on measures:
Not known

8.22 Avoid mirror fragments in class structures

Do not use duplicate structures. Try to factor out the common structures.

Definitions:
A *mirror fragment* is a fragment with the same structure as another fragment (the original fragment). For every class in the original, a class with the same structure exists in the mirror fragment. Two classes have the same structure, all of their attributes and all of their operations have the same names, and they have the same relationships to other classes.

Rationale:
Mirror fragments represent duplicated functionality within a schema. If schema changes are required, they have to be repeatedly performed in all the places where the functionality is redundantly defined. This raises the effort of adapting and maintaining the schema.

Position in Life Cycle:
Analysis, Design, Coding

Granularity:
Classes, operations, attributes

Example:
violating:

```
class Contract{}
class Address{}
class Customer{
        integer customerNumber;
        float validation;
        Contracts theContracts;
        Address theAddress;
        void assignContract(Contract aContract)
}
class Company{
        integer numberOfEmployees;
        set(Customer) customers;
        Contracts theContracts;
        Address theAddress;
        void assignContract(Contract aContract)
}
```

Complying:

```
class Contract{}
class Address{}
class Party{
        Contracts theContracts;
        Address theAddress;
        void assignContract(Contract aContract)
}
class Customer: Party{
        integer customerNumber;
        float validation;
}
class Company: Party{
        integer numberOfEmployees;
        set(Customer) customers;
}
```

Subsuming/subsumed heuristic:
Not known

Checking rules:
Check for structural equivalence (recursive):

Check if each two classes have the same structure. If two classes with the same structure are found, insert them in the result fragment. Check for each two classes related to both classes if they are structurally equivalent.

Transformation rules:
Define base classes for all pairs of structural equivalent classes. Move the duplicated attributes and operations into the base classes.

Violated heuristics:
Not known

Justifications for violating the heuristic:
Not known

Effects on measures:
Size: positive

8.23 Dependencies in inheritance hierarchies should not go from higher to lower levels

Do not define dependencies from base classes to bottom classes in the same or in other inheritance hierarchies.

Rationale:
Dependencies from soft-coupled classes:
The higher a class is positioned in an inheritance hierarchy, the more classes (all of its children) depend on this class. This means that higher classes are coupling-harder than lower classes. Coupling-hard classes should not depend on coupling soft classes.

Dependencies from low-level detail:
High-level classes should not depend on low-level details. The Dependency Inversion Principle (see (Martin 1995a)) states that both high-level policy and low-level details should depend upon abstractions.

Position in Life Cycle:
Analysis, Design, Coding

Granularity:

Example:
Violating:


```
class Person{
        set(Customer) myCustomers;
        set(Customer) myPrivateCuzstomers;
static create(string PersonType, int id){
switch PersonType:
        case Customer: myCustomers.add new Customer(id);
        case PrivateCustomer: myPrivateCustomers.add new PrivateCusto-
mer(id);
}
}
class Customer: Person {
}
class privateCustomer: Customer{
}
```

Complying:

```
class PersonFactory{
set(Customer) myCustomers;
set(Customer) myPrivateCustomers;
static create(string PersonType, int id){
switch PersonType:
        case Customer: myCustomers.add new Customer(id);
        case PrivateCustomer: myPrivateCustomers.add new PrivateCusto-
mer(id);
}
}
class Person{
}
class Customer: Person {
}
class privateCustomer: Customer{
}
```

Subsuming/subsumed heuristic:
Subsuming: Hard fragments should not depend on soft fragments.

Checking rules:
Calculate depth in inheritance tree (DIT) for all classes. Check every dependency between a client- and a server class for *DIT(clientClass)>DIT(serverClass)*.

Transformation rules:
Define another abstraction for the base class which encapsulates the knowledge of what the derived class depends on. Move the dependency to this new abstraction (see example). Move the dependencies to the common base class.

Violated heuristics:
Not known

Justifications for violating the heuristic:
Dependencies from exception classes, e.g., in Java (Gosling, Joy et al. 1996; Gosling, Joy et al. 2000).

Effects on measures:
Not known

8.24 Hard fragments should not depend on soft fragments

Do not define dependencies of coupling-hard fragments on coupling-soft fragments. Soft fragments may depend on hard or soft fragments, and hard fragments should only depend on each other.

Rationale:
In most cases, a fragment is coupling-hard because the effort to change the fragment would be high. Changes in a soft fragment often occur by definition. Changes in any fragment tend to propagate to the dependent fragments. This means that the soft fragments are a source of change for the dependent hard fragment. Changes in a coupling-hard fragment require a high effort.

Position in Life Cycle:
Analysis, Design, Coding

Example:
Violating:

```
class Customer{
        Order myOrder;
}
class CustomerDialog{
        Customer myCustomer;
}
class CustomerValidator{
        Customer myCustomer;
}
class CustomerReport{
        Customer myCustomer;
}
class Order{
        set of Transaction myTransactions;
        OrderWorkflow myWorkflow;
        OrderBook myOrderBook;
}
class Transaction{
}
class OrderWorkflow{
}
class  OrderBook{
}
```


Complying:

```
class CustomerOrderDirector{
        Customer myCustomer;
        Order myOrder;
}
class Customer{
}
class CustomerDialog{
        Customer myCustomer;
}
class CustomerValidator{
        Customer myCustomer;
}
class CustomerReport{
        Customer myCustomer;
}
class Order{
        set of Transaction myTransactions;
        OrderWorkflow myWorkflow;
        OrderBook myOrderBook;
}
class Transaction{
}
class OrderWorkflow{
}
class  OrderBook{
}
```

Subsuming/subsumed heuristic:
Subsumed:
8.23 Dependencies in inheritance hierarchies should not go from higher to lower levels.
8.15 Soft classes should not be base classes.

Checking rules:

Calculate the coupling hardness of every fragment. Check every dependency between two fragments f1, f2. If HF(f1)>HF(f2) (see section 7.9), the heuristic is violated.

Transformation rules:

Use inheritance in order to define new abstractions for the "server" fragment. Relocate the dependencies from the server to the new abstractions.

Violated heuristics:

Not known

Justifications for violating the heuristic:

Not known

Effects on measures:

Not known

8.25 Avoid direct recursive associations

Avoid defining an association from a class to itself.

Definitions:

Direct recursive association: A direct recursive association is an association in which source and target class are the same.

Rationale:

Recursive associations are frequently used to model linked lists or tree structures of objects. These structures are often key elements of an application. If a direct recursive association is used, it is difficult to add new types of nodes in the linked list or the tree structure. Besides this, the class with the direct recursive association has to play two different roles: the role of a container and the role of the content. This often leads to situations in which two different key abstractions are modeled within one class.

Position in Life Cycle:

Analysis, Design, Coding

Granularity:

Classes, associations

Example:

Violating:

```
class Whole{
        int numberOfParts;
        int partNumber;
        set(Whole) myParts;
}
```


Complying:

```
class Whole{
        int numberOfParts;
}
class Part: Whole{
        int partNumber;
        set(Whole) myParts;
}
```

Subsuming/subsumed heuristic:

Not known.

Checking rules:

Check all associations of a class. If an association is found that references the class, the heuristic is violated.

Transformation rules:

Composite transformation: The class with the direct recursive relation is replaced by a composite (Gamma, Helm et al. 1994).

Let x be the class with a set of direct recursive associations.
Insert new class *abstractX*.
Insert inheritance relation from *x* to *abstractX*.
Make abstract *x* the new target of all associations in the set.

Violated heuristics:

Not known

Justifications for violating the heuristic:

Not known

Effects on measures:

Not known

9 Objectbase Design with MeTHOOD

In section 4.2, we have proposed the following requirements for a "good" objectbase:

r1. It should provide effective support for shared objects with explicit control of lifetime and independence of creation and use.
r2. It should contain a comprehensible (standardized) and simple abstraction of (complex) entities of the real world.
r3. The state of the objectbase should be consistent with real world integrity constraints, its behavior should enforce these constraints.
r4. It should be free of anomalies and redundancies.
r5. It should be changeable and extensible with little effort.
r6. It should be independent of other programs, shared and (re-) usable.

The following sections show how MeTHOOD helps designers to achieve these requirements.

MeTHOOD measures, heuristics and transformation rules applied for objectbase design.

9.1 Support for shared objects

An objectbase schema should provide effective support for shared objects, with explicit control of lifetime and independence of creation and use.

Objects in an objectbase have an explicitly controlled lifetime. If we consider current systems and all used data migration tools, we observe that most objects in an objectbase live longer than the management system they are initially created in. Since the schema of an objectbase describes the structure of an objectbase, it should also have a long lifetime. If the schema is specified in the schema definition language of the objectbase, it depends on the used (potentially short living) objectbase management system. This is one of the reasons for the so-called *conceptual-logical schema separation* in relational database design. The most common way to obtain a logical *database* schema is to design a conceptual database schema and map it to the logical schema (see section 3.3.2).

The conceptual logical separation introduced for relational database design is needed in objectbase design, too.

We propose the same separation between conceptual and logical schema of objectbase design. Designers should first specify a *conceptual objectbase schema* and map this schema to the *logical objectbase schema* (see Figure 19).

Objects in an objectbase are persistent representations of real world entities. This has a strong influence on the design of their classes. In a good design, this influence is taken into account: On the one hand, the objects are persistent. This raises the effort required for certain schema modifications (e.g. moving an attribute from one class to another). This is the case for good schema modification facilities in the OBMS. Some current OBMSs do not even have such facilities. Even if the OBMS provides powerful schema modification facilities, changes are hard to do. Consider a class *privateCustomer* (with thousands of existing objects) in which we need to add an additional attribute *hobby*. The required information has to be collected in order to execute the schema change.

On the other hand, the objects represent real world entities. This raises the potential for changes in their class. This is because changed business requirements often change the information to be provided by an object. Consider, for example, the real world entity *customer* (of a bank, airline, insurance,...). In most business areas, changing requirements lead to changes in what information is captured of a customer. If the information represented in an object changes, in most cases, the class has to be changed. We have observed that nearly all changes (from real world entities) are newly added elements, e.g., new attributes or new classes.

This means that on the first look, the objectbase raises the potential for changes in the schema and makes required changes expensive. Such a situation is not at all desired by designers. A closer look at the notion of *stability of a class* and the types of required schema changes will bring us to another situation.

Stability of a class

Something is considered stable if the probability that it will change is low. If this probability is high, it is considered instable. The *probability of modifications* is one of the most fundamental considerations in design (not only in software design!). In building architecture, for good reasons much effort is spent for calculating the stability of buildings (a completely different type of stability with similar effects as those described here) and their parts.

The probability of modification is not enough considered in most current object-oriented design approaches. One of the reasons for this is that the effects of badly balanced stable and instable software artifacts are recognized late in the development cycle and hard to trace back to specific design

decisions. The approaches presented in section 11.3.2 and 11.5.3 consider modification probability explicitly. (Martin 1995a) defines stability (Martin Stability) as "the probable change rate" and makes some important observations: Stable does not mean bad. In some cases, stability is highly desirable. In most cases, a software artifact is stable because the effort to change it is high. An objectbase schema has two main sources of stability: its *persistent objects* and its *client applications*.

On one side, the effort to change (add new attributes and change values of existing attributes) existing persistent objects is high. In most cases, extra programs have to be written. In some cases, even worse operational measures have to be taken. On the other side, an objectbase always has client applications. If the schema of the objectbase is changed, in many cases, the clients have to be cahnged, too. *This means that objectbases primarily have a stabilizing effect on their schemas.*

Stable fragments are bad if they have to be changed frequently (because these changes are expensive). This kind of non-desirable stability is called *rigidness* by (Martin 1995a). We refer to this situation as *coupling tension. Some of the most important tasks of an objectbase schema designer are to*

- treat classes with persistent objects as stable,
- raise coupling hardness and
- avoid high coupling tension in these classes.

To support designers with this task, the MeTHOOD catalogue contains a set of measures that assess change probabilities. They allow designers to assess the *coupling hardness* of fragments. Coupling hardness is an important factor to determine stability based on dependencies. Using these measures, designers can evaluate whether classes with persistent objects are as coupling hard as they should. Furthermore, we provide many heuristics and measures that consider tensions in the schema. A top level heuristic in this area is "Hard fragments should not depend on soft fragments." which is adapted from Martins 8th design principle in section 11.3.2: "Never cause a stable category to depend upon a less stable category."[18] Other heuristics are "Avoid dependencies of objectbase classes on their clients" and "Dependencies in inheritance hierarchies should not go from higher to lower levels"

Breaking inheritance relationships

Persistent objects have a further influence on the structure of the schema. Since their classes represent mini world concepts, new attributes are often added (an effect of changing requirements). A class under such an influence is a bad candidate for a base class. We provide a heuristic for this problem ("Soft classes should not be base classes"). The fear is that the inheritance

[18] The heuristic has been changed from the original because it is closer to the core of the statement.

relation breaks because the *meaning* of the base class changes. For example, we often observe a situation in that a class *Customer* "shifts" to a *Company* class. Deriving *PrivateCustomer* from *Customer* initially makes sense, but the shift breaks the golden rule that the inheritance relationship is reasonable, e.g., a *PrivateCustomer* is a customer, but a *PrivateCustomer* is not a company.

Performance considerations

Performance issues always have an influence on design. Accessing an object in an objectbase is in general a slower process than accessing other objects. However, performance should mainly be considered during implementation, and the conceptual schema should not be affected too strongly. We have observed one case in which performance optimizations are useful in conceptual design. This case is captured in the heuristic "Avoid contained objects that can concurrently be modified", which is considered in section 8.12.

9.2 Comprehensible real world abstraction

An objectbase schema should contain a comprehensible (standardized) and simple abstraction of (complex) entities of the real world.

The schema of a database must be understood by many different actors, as described by (Elmasri, Navathe 1989). The same is true for objectbases. There are analysts, end users, objectbase designers, application programmers and tool developers – they work in different projects, sometimes even for different customers, and all of them have to comprehend and agree on the same objectbase schema. This is probably the most fundamental challenge for objectbase designers. Unfortunately, this area is not thoroughly understood, and there is very little support for this task. MeTHOOD only provides very basic help.

In many recent projects, we have made the experience that model walkthroughs are an efficient way to give a group of people a basic common understanding of a set of interrelated classes. This type of walkthrough is proposed in (Graham 1993). It is based on class cards. Attendants of a walkthrough simulate objects. If we consider the class card in Table 11,one attendant could simulate a *ProductFactory* object and another a *Product* object. The walkthrough could start with the question "What happens if we create a new product?" which would be the responsibility *captureProduct* of the *ProductFactory* "simulator". In the collaborators section of this responsibility, an attendant would see what has to be done. He would create a new *Product* (*new* is an implicit operation of every class). The creation message would "wake up" the *Product* simulator, who would get the message *init* im-

mediately afterwards. The attendant could then consider the *init* responsibility on his class card to see what would happen next.

Table 11: Example class cards

Class: Product Factory	Baseclasses:
Attributes	
Public Operations Product captureProduct ()	Collaborators Product new Product init
Private Operations	Collaborators

Class: Product	Baseclasses:
Attributes	
Public Operations void init ()	Collaborators
Private Operations	Collaborators

We propose to use these walkthroughs for the design of objectbases. The MeTHOOD metamodel (see section 2.3) supports the specification of class cards – all language elements used on class cards are available in the MeTHOOD model. Furthermore, all proposed measures support the level of abstraction found in class cards.

As we have seen, walkthroughs can help different actors to get a *common* understanding of the schema of an objectbase. An important prerequisite to comprehend such a schema at all is to see that it is simple. The main objective of the size measure we propose (see sections 7.1 and 7.5) is to help designers understand how design decisions influence the simplicity of the schema, which is required for objectbase design.

Another key feature of a good objectbase schema, which is comprehensible to a large group of different actors is that *domain names* are used for all schema elements (attribute, class, operation,...). A domain name is a term that has already a defined meaning[19] for most experts of the domain to be modeled. Especially, names of classes should be domain names. We call a class with a domain name a *key abstraction*. A good objectbase schema only consists of key abstractions. A flaw found in many objectbase schemas are classes which can be split into different key abstractions. Another flaw which often occurs is that different classes are defined which should be defined in one key abstraction. Again, we recommend walkthroughs (especially with domain experts) to eliminate these kinds of flaws. Finally, we propose the heuristics in section 8.5, 8.10, 8.9 and 8.14. to detect them.

[19] Often, it is very helpful to use only terms defined in a domain specific dictionary, which is available in most business domains that we know.

9.3 Enforcement of consistent objectbase state

An objectbase schema should enforce a consistent state of the underlying objectbase: the objectbase should be consistent with real world integrity constraints.

A good schema should not allows data redundancies

The state of an objectbase is the set of all values in the objectbase. The most essential requirement to this state is that it adequately represents real world facts. This is particularly important if the objectbase is used as the "master record" of these facts. Consider, for example, the current amount of money on your bank account. This requirement is important to all levels of the organization that owns the objectbase, because it is a prerequisite for the trust of their customers.

Therefore, every organization must take measures to assure the integrity of their objectbases. One objective of objectbase design is to reduce the costs for these measures. Often, *objectbase owners* are assigned to specific objectbases to execute the necessary activities. The owner has to give guarantees about the state of the objectbase. These guarantees represent a real value and require investments. A good objectbase design makes it *possible to give the necessary guarantees* and assures that *the investment for the guarantees is not higher than required.*

Consider an objectbase schema containing a class *Account* (with some information about the owner) and a public attribute *amount*. The most essential integrity constraint for accounts is that every change of the amount is reflected in the inverse change of the amount in another account. It assures that money does not simply appear or vanish. The owner of an objectbase with the class *account* described above is not able to give any guarantees about the values of the attribute *amount* (because *amount* is public). The only way to ensure this integrity constraint is to control every location of every client who uses it, which is very ineffective and expensive. Public attributes in the schema of an objectbase avoid guarantees and make clients responsible to control and enforce integrity. We do not even consider the resulting code redundancies within the different applications, which would result in additional effort.

To allow more effective guarantees, the schema of an objectbase must have a special form. In this form, *only special operations* (see below) are visible. If the schema has this form, it is possible to specify and enforce integrity constraints on the schema, implement these integrity constraints (using an

available objectbase system) and enforce them at runtime. The specification of integrity constraints can be done with approaches like (Ceri, Fraternali 1997), or active rules (Dayal, Buchmann et al. 1988; Gatziu, Dittrich 1993; Ceri, Fraternali 1997), or even state diagrams (OMG 1997b).

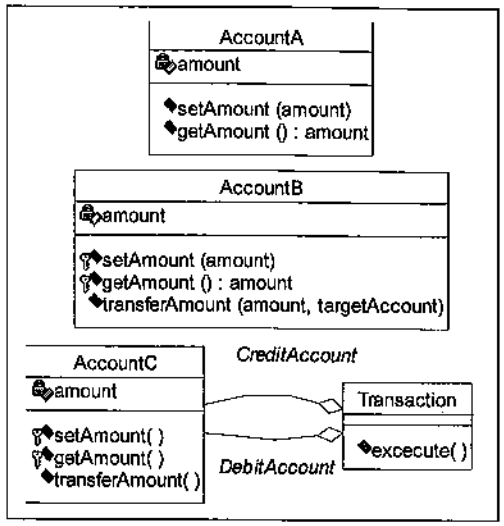


Figure 37: Transactions and information hiding

MeTHOOD also considers the form of the schema. One quality aspect of a schema is that it allows effective guarantees. MeTHOOD provides measures and heuristics that support designers to improve this quality aspect. For example, compliance with the heuristic "Every attribute should be hidden within its class" results in basic encapsulation of attributes. Encapsulation means that only operations are available to the client. For our discussion above, this is not enough. *Information hiding* is needed.

Information hiding as defined in (Kilov, Ross 1994)[20] means that only operations that preserve information correctness are available to the client. Information hiding implies encapsulation, but encapsulation does not imply information hiding. Consider the concept of a financial transaction. A transaction is a transfer action in which an amount of money is transferred from one account to another. The essential integrity rule of a financial transaction is that the sum of the transfer is zero, e.g., 100$ are debited to account 345 and credited to account 234.

[20] This definition of the term information hiding is stronger than the original "Parnas definition", therefore, it is used here.

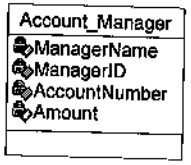
Consider the designs of the accounts in Figure 37. *AccountA* provides encapsulation, but not information hiding. The owner of *AccountA* cannot guarantee that the "transaction integrity rule" holds. *AccountB* and *AccountC* provide information hiding. Clients of *AccountB* and *AccountC* cannot use the setter operation *setAmount*. *AccountB* and *AccountC* only provide operations to their clients that preserve information correctness. *AccountC* and *Transaction* allow capturing transactions explicitly, e.g. to record transactions in a history.

The heuristic "Avoid pure accessor operations" helps designers to go further in the direction of information hiding. Furthermore, the MeTHOOD encapsulation measures give designers a good hint of how good a schema is (from the point of view of information hiding). A more complete advice for information hiding is contained in the rule: "Only operations that preserve information correctness should be available to the client.", which is hard to check because it is impossible to specify which information is correct. This rule specifies the line between operations in an objectbase and operations in the clients of the objectbase.

9.4 Absence of anomalies and redundancies

An objectbase schema should be free of anomalies and redundancies.

Another important integrity aspect of objectbases are wrong entries made by objectbase users, or wrong entries which occur because objectbase clients or operations in the objectbase itself are error prone.



ManagerName	ManagerID	AccountNumber	Amount
Smith	39874	23876-3498	234234
Smith	39874	23812-2378	324324
Doe	34759	23812-2378	324324
Doe	34759	23812-2658	343243

Figure 38: CustomerAccount

It is impossible to consider all possible sources of these errors, but we can identify a class of error sources which can be avoided. These redundancies

and anomalies are well-known from relational database design. Consider the schema fragment in Figure 38. To create a new *AccountManager* object, we have to add the *Account* information too, even if the manager does not yet manage any accounts – or we have to use null or default values. To add a new account which is not yet managed by a manager, we have to set null or default values. The resulting anomalies are called *creation anomalies*.

Furthermore, if we delete the last account of a manager, the manager is also deleted. This type of anomaly is called *deletion anomaly*. The last type of anomalies considers modifications of attribute values. If we need to change the name of a manager, we have to change it in many different objects. Should we overlook any objects, the objectbase gets inconsistent. This type of anomaly is called *modification anomaly*.

This last type of anomaly can occur if the objectbase contains redundancies. A redundancy means that the same information occurs more than one time in the objectbase. If this is the case, it can happen that the information is changed in only one of the locations in which it occurs, which means that the information becomes inconsistent.

The approaches to resolve these types of anomalies in relational database design are to create relations which comply with certain normal forms (see section 3.3.1 Normal forms and algorithmic database design). (Kilov, Ross 1994) claims that these design techniques are no longer needed in object modeling. OMS vendors even claim that encapsulation in object-oriented design eliminates the need for normal forms and transformations.

In our approach, some ideas from relational normal forms still play a role. The first normal form (1nf) is no longer needed, but we propose two heuristics, that revitalize the ideas of the third normal form (3nf) and avoid multi-valued dependencies (see section 8.5, 8.9, 8.10). We have two arguments for this type of normalization of an object-oriented schema. Figure 38 shows that object-oriented modeling languages still allow to define classes, which can imply major anomalies. The difference to relational databases is that OBMSs do not in general provide generic update operations like insert, update and delete. Instead of this, user defined object constructors and operations for object manipulation can be used. This means that it is possible to fence potential anomalies to the operations of a single class. However, it requires additional effort to define the operations which encapsulate the class properly. Furthermore, the operations of the considered class can still have errors which result in the potential anomalies in the class. The second argument for the proposed normalization is "real world modeling". We have observed that schemas that reflect the real world adequately do not in general include classes that imply anomalies. It seems that "real world" objects simply do not show these types of anomalies. Thus, if we detect a location in a schema in that anomalies are implied, we consider this location as a potential design flaw.

This kind of normalization is closely related to the concept of *cohesion*. Cohesion is a term describing the inner relatedness of a class to itself. A highly cohesive class represents one key abstraction, and all properties of this class are related to each other. If a class is not normalized, it often represents more than one key abstraction. The class in Figure 38 represents customer-managers and accounts. In this situation, we usually get different groups of operations which use different groups of attributes, and the considered class lacks cohesion. Our proposed cohesion measure (see section 7.11) assesses this type of cohesion. The proposed heuristics (see sections 8.5, 8.9, 8.10) detect locations in which normalization should be applied in order help to improve the measure.

9.5 Extensibility

An objectbase schema should be changeable and extensible with little effort.

It is often necessary to change the schema of an objectbase. One advantage of traditional DBMSs, in comparison to existing OMSs, is that the traditional systems provide techniques that allow modifying the schema dynamically at runtime. It is, for example, possible to add and delete attributes and tables. Support for these features is very limited in current OMSs, and if schema changes are required, the system has to be stopped, modified, in most cases recompiled and started. Therefore, flexibility and changeability need special attention during objectbase schema design.

These changes are expensive if the schema is rigid, and therefore, one of the main tasks of designers is to avoid this rigidness (as we have already seen in section 9.1.). In order to examine what makes a schema rigid, we need to take a closer look at *the sources for change* and the *sources for stability*. Such a source is an entity which has influence on schema fragments. Examples of sources are other fragments, users and other programs. We distinguish between schema internal and external stability and instability sources. Schema internal sources are present in the schema and under the control of the designer, schema external sources are outside the schema and, in general, not under the control of the designer.

Instability through changes in dependent classes.

Examples of internal stability sources are other schema fragments, which depend on a supplier schema (dependent schema fragments). These dependencies make the supplier fragment coupling-hard, because changes tend to propagate, and the dependents have to be changed, too. This raises the effort for the change. On the other hand, dependencies from other fragments are a schema-internal source of instability. Examples of external stability are persistent objects (as already discussed) and other dependent programs. Sources of external instability are changes in user requirements and changes in product interfaces.

Table 12: Sources of stability and instability

Schema	Internal	External
Stability	Dependent schema fragments	Persistent objects Dependent programs
Instability	Dependencies from other schema fragments	Changing user requirements, Changing product interfaces

To achieve a changeable and flexible schema, *the designer has to adapt the structure of the schema to the different sources of stability and instability*. In this section, we show how MeTHOOD supports this task for *internal* stability and instability sources. The next section provides advice for the external sources.

Schema adaptations for balanced stability

To adapt the schema to internal sources of stability and instability, the designer must consider the classes within the schema. He must

1. have a good understanding about which fragments are stable and which are instable,
2. find locations with non-desirable dependencies (e.g., where coupling-hard fragments depend on soft fragments) and
3. shield fragments from each other by relaxing the non-desirable dependencies.

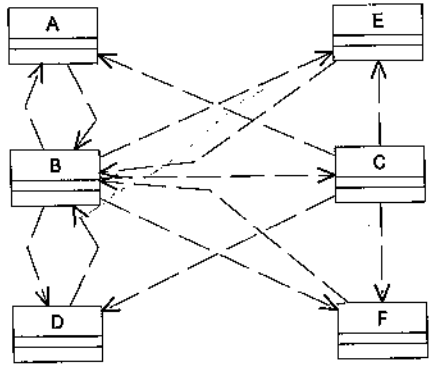


Figure 39: Dependent Classes

MeTHOOD supports 1. by measures (e.g. coupling hardness) considered in section 7. Look at the schema fragment in Figure 39. It has been extracted from a real system, and simplified. One of the problems of this schema is

that the changes tend to propagate through the schema, e.g., if we add an attribute to an operation in D, we often have to make subsequent changes in B and A.

We can calculate the coupling hardness and tension of classes in Figure 39 according to the MeTHOOD coupling hardness measure (see section 7.9). Assuming that all dependencies have the strenght 1 we get the following figures:

Table 13: Coupling hardness and tension of classes

	Coupling hardness	Coupling tension
A	1	2
B	-1	8
C	-3	2
D	1	2
E	1	2
F	1	2

We see that the values represent our expectations: the classes A, D, E and F have the same dependencies, and they should be coupling-hard, because each of them depends on two classes and only one class depends on each of the classes. The classes B and C are coupling-soft. C is softer than B, because fewer classes depend on B. Now, we have a good overview of the coupling hardness of these classes. But what do we have to change?

MeTHOOD provides a number of heuristics, that detect different kinds of potential flaws in this specific fragment (supporting 2.).

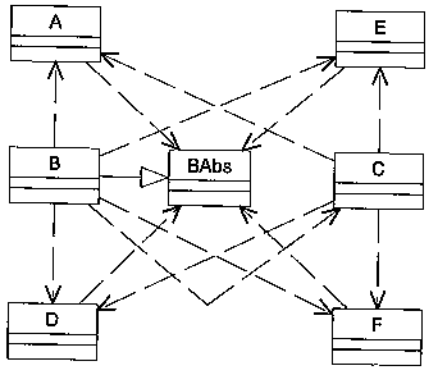


Figure 40: "Relaxed" Classes

One example of such a heuristic is "Hard fragments should not depend on soft fragments", which is violated four times in this fragment, since the

classes A, D, E, F depend on B, and B depends on C. The heuristic also points to a (quite generic) transformation rule (supporting 3). The transformation rule suggests inserting a new abstraction for B and redirecting the dependencies from B to the new abstraction (see Figure 40). The resulting fragment does not violate this heuristic[21] and has the following measures.

Table 14: Coupling hardness and tension

	Coupling hardness	Coupling tension
A	1	2
BAbs	5	0
B	-5	0
C	-3	2
D	1	2
E	1	2
F	1	2

B is now soft, and nothing depends on it. We can now make changes in B and guarantee that none of these changes propagate. A new class *BAbs* has been introduced, which is very hard. *Coupling-hard classes do not depend on soft ones.* We can also observe another related improvement of the fragment. We have shown measures for the coupling tension (see section 7.10) of the classes in the fragment. This gives hints about the probability that changes propagate through the fragment. In the original fragment, the class with the worst coupling tension was B, which now has been improved with the transformation. In the "real" schema, we were able to properly name the new abstraction (which is very important and should be checked for every transformation).

This example shows how designers are supported in understanding internal dependencies and coupling hardness of an objectbase schema and improving changeability of fragments. The MeTHOOD approach provides different heuristics that propose various kinds of structural changes in a schema.

Configuring design information at runtime.

The knowledge layer

A knowledge layer is a schema fragment that is populated with objects at configuration time of the objectbase. Such a schema fragment allows adding information at run-time, which is otherwise often statically modeled, making an objectbase more flexible. The knowledge layer has mainly been proposed for analysis (Hay 1995; Fowler 1996), but we have efficiently introduced knowledge layers during design.

[21] Other heuristics are still violated. The transformation did not introduce new violations.

Consider the problem of creating an objectbase schema for a report writer, a schema that can provide different kinds of reports for a business. It should be possible to combine sub-reports of existing reports (e.g., the success report as part of a business report) to new reports, and it should be easy to add new kinds of reports. The first solution that comes to mind would be a composite pattern (Gamma, Helm et al. 1994) style schema like the schema in Figure 41.

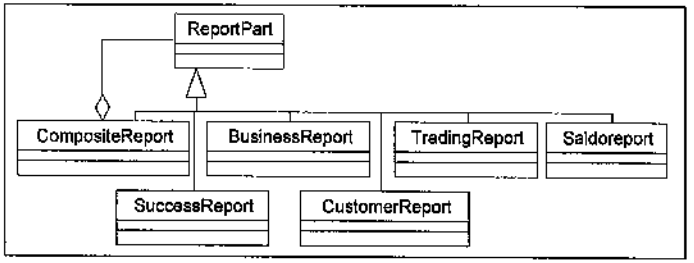


Figure 41: Composite report

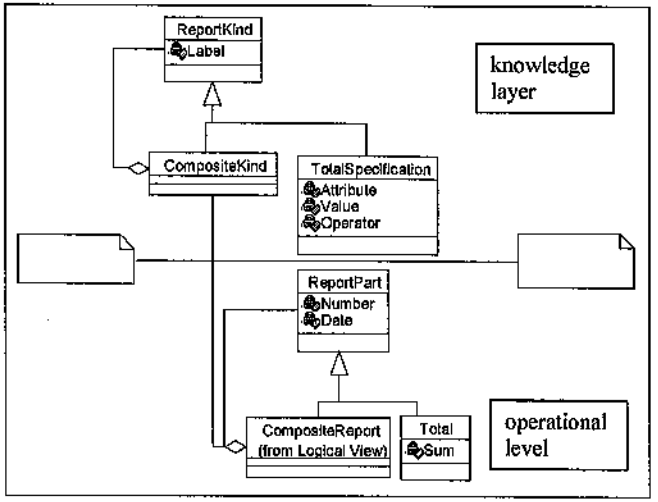


Figure 42: Schema with knowledge layer

Such a schema allows combining existing reports with new reports. However, adding a new type of report requires schema changes, testing and even changes in the client programs. A schema using a knowledge layer could look like the schema in Figure 42.

The upper part of the schema is the *knowledge layer*. It is configurable at configuration time or at run time. Here, users can create new types of reports (using *ReportKind* and *CompositeKind*) and assign access strategies to gather the data required in the report (using *TotalSpecification*). The lower level of the schema is the *operational level*. This level is used to create the different reports based on the specification of the report types.

This type of knowledge/operational separation is found in almost every banking or insurance system. A similar structure to the structure above can be used to model, e.g., different kinds of products or contracts. The example above shows how classes can be transformed to objects if flexibility is required. The knowledge layer can also be used to dynamically add attributes and operations. We recommend considering the knowledge layer approach if the number of classes, attributes and operations is very high and these elements have to be changed and reconfigured frequently.

9.6 Schema - program independence

An objectbase schema should be independent of other programs, and (re) usable.

In the above section, we have considered the internal relationships of a schema. Now, we consider the external relationships of an objectbase to its applications. Objectbases should be efficiently *shared* by different applications. Shared means that the applications can use the same parts of the objectbase at the same time. Efficiently means that the *sharing costs* should be low. The sharing costs are the development and maintenance costs of the applications and the objectbase, that arise from sharing the objectbase.

The sharing conflict

Sharing results in inner conflicts of classes in the objectbase. If such a class is shared, there exist dependent client applications. These applications are sources for change requests. This is because change requests to the applications themselves often propagate to the objectbase. This means that the applications are sources of instability for the objectbase schema. On the other hand, they depend on the objectbase schema and have to be changed if the objectbase schema changes.

The conflict is that classes in the objectbase should be stable (because many applications depend on it and it represents persistent objects). The same applications can make it instable. This conflict is illustrated in Figure 43. It is one of the reasons that designing the schema of an objectbase is very hard.

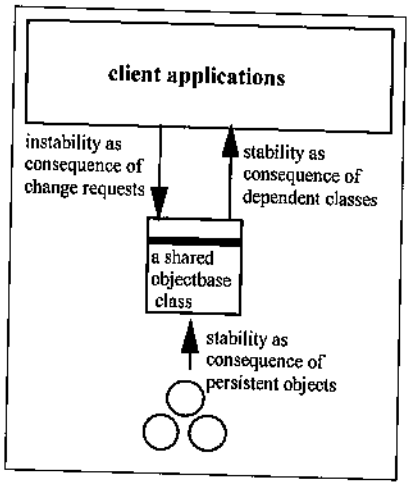


Figure 43: Shared objectbase class

The objectbase schemas tend to get *rigid* very easily (changes are required, but cannot be executed). Besides this, in most cases, the shared fragments belong neither logically nor organizationally to the application project groups. It is owned and controlled by others, and often, it is physically separated from the client. This also has an impact on design, because the users of the fragments cannot control changes in the fragments.
The task of the objectbase designer is to assure that schema changes which propagate to applications, do not often occur.

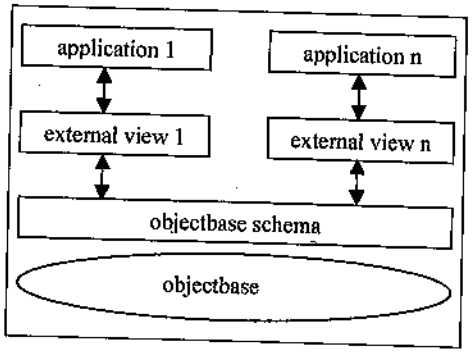


Figure 44: View architecture

Another problem closely related to the problem above is that changes in the objectbase schema must be traceable to the influenced applications. *Traceability of influenced applications* means that if a fragment of the schema is

changed, it is easy to find each location in each application in that subsequent changes are required.

Traceability is extremely important, because a schema in which changes cannot be traced is very rigid (because all dependent applications have to be checked). This often leads to a situation in which only additions in the schema are allowed (in most cases they do not influence existing applications), and modifications and deletions of schema elements are forbidden.

For relational databases, a special schema architecture (e.g. (Elmasri, Navathe 1989)) has been proposed (see Figure 44). This schema architecture includes separate view definitions for different applications. The different applications use the view only to access the database schema. This schema architecture assures that few changes in the database schema structure propagate to the applications (many changes require only changes in the view definitions) and that the changes are traceable (if the structure of a specific application view changes, only this application may require subsequent changes).

The objectbase has to provide similar mechanisms to assure *effective* sharing. The designer of an objectbase has to

- provide different views,
- minimize dependencies of client applications of the schema ,
- localize these dependencies,
- recognize external instability,
- shield stable schema fragments from external instability and
- treat external stable fragments as coupling-hard fragments.

MeTHOOD provides different forms of support for these problems. The notion of *design fragments* in MeTHOOD is a prerequisite for efficient support in these tasks. Designers can specifically select the parts of the schema and the applications that they need to consider, measure and improve.

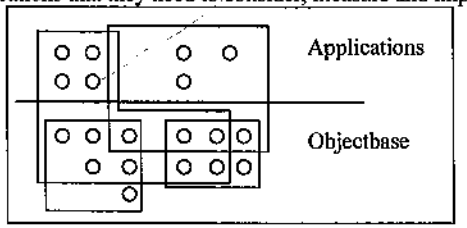


Figure 45: Fragment topology

Consider the topology in Figure 45 (the areas denote fragments, and the circles denote classes). It consists of an objectbase schema with two separate fragments. It is used by two applications which share the fragments of

the objectbase. The MeTHOOD fragment concepts allow designers to select a set of fragments and evaluate this selection. E.g., if a designer has to measure how strong both applications depend on the whole objectbase, he can define all classes in both modules as a fragment and all classes in the applications as a second fragment and then measure the dependencies between both fragments. The same applies to all other measures and all heuristics.

The second form of support are the coupling measures and different related heuristics. These measures are specifically designed for the problems related to coupling of applications with an objectbase. The heuristics "Avoid dependencies of objectbase classes on their clients", "Avoid contained objects that can concurrently be modified", "Soft classes should not be base classes", "An operation should only use classes of attributes of its class, classes of its parameters or classes of locally created objects", "Dependencies in inheritance hierarchies should not go from higher to lower levels", "Hard fragments should not depend on soft fragments" with their transformation rules provide further support for this task. Consider section 8.1 for specific information about these heuristics. It is important to note that sharing has to be effective, i.e. the development and maintenance cost resulting from sharing has to be reduced.

Reducing application development and maintenance costs

A core argument for using an objectbase vs. a traditional database as a central shared component is that this approach can reduce the development and maintenance costs of the applications of the objectbase. Development cost can be reduced if data *and functionality* developed for one application (and provided as a part of the objectbase) can be reused in other applications. Sharing can reduce maintenance costs, because the shared sources (of the objectbase) exist only once. Redundancies, that often occur with so called *fat client applications* accessing a traditional database, can be efficiently eliminated.

It is important to note that these types of cost reductions are not automatically achieved when an objectbase is used. Again, design of the objectbase plays the most important role realizing this potential. It is particularly important to "find a good line of separation" between the objectbase and its applications. Again, the hiding factor measures and the heuristics "Every attribute should be hidden within its class", "A class should capture one, and only one key abstraction with all its information and its entire behavior" and "Avoid pure accessor operations" support designers in realizing a schema for effective sharing.

9.7 Conclusion

In this section, we have considered some of the main requirements for an objectbase. We have illustrated that these requirements are not automati-

cally fulfilled if an OBMS is employed, and that a well-designed objectbase schema is one of the most important prerequisites to fulfill these requirements.

Table 15: MeTHOOD support for objectbase requirements

Objectbase requirement	Measures/Heuristics
r1. It should provide effective support for shared objects with explicit control of lifetime and independence of creation and use	*Measures*: Coupling hardness of a fragment (HF) Coupling tension of a fragment (TF) *Heuristics*: "Hard fragments should not depend on soft fragments." "Avoid dependencies of objectbase classes on their clients" "Avoid contained objects that can concurrently be modified" "Soft classes should not be base classes" "Avoid contained objects that can concurrently be modified"
r2. It should contain a comprehensible (standardized) and simple abstraction of (complex) entities of the real world	*Measures*: Size of design fragment (SF) Public factor of design fragment (PF) Cohesion of a fragment (COF) *Heuristics*: "A class should capture one, and only one key abstraction with all its information and its entire behavior" "Avoid multivalued dependencies" "Avoid classes with properties implying redundancies" "Common properties of objects should be defined in a single location"
r3. The state of the objectbase should be consistent with real world integrity constraints, its behavior should enforce these constraints.	*Measures:* Size of design fragment (SF) Public factor of design fragment (PF) *Heuristics:* "Avoid pure accessor operations"

Objectbase requirement	Measures/Heuristics
r4. It should be free of anomalies and redundancies	*Measures:* Size of design fragment (SF) Cohesion of a fragment (COF) *Heuristics:* "A class should capture one, and only one key abstraction with all its information and its entire behavior" "Avoid classes with properties implying redundancies" "Avoid multivalued dependencies"
r5. It should be changeable and extensible with little effort	*Measures:* Coupling of fragment (CF) Inverse coupling of a fragment (ICF) Coupling hardness of a fragment (HF) Coupling tension of a fragment (TF) *Heuristics:* Different dependency heuristics, e.g., "Hard fragments should not depend on soft fragments"
r6. It should be independent of other programs, shared and (re-) usable	*Measures:* Inverse coupling of a fragment (ICF) Coupling hardness of a fragment (HF) Coupling tension of a fragment (TF) *Heuristics:* "Avoid dependencies of objectbase classes on their clients" "Soft classes should not be base classes" "An operation should only use classes of attributes of its class, classes of its parameters or classes of locally created objects" "Dependencies in inheritance hierarchies should not go from higher to lower levels" "Hard fragments should not depend on soft fragments" "Every attribute should be hidden within its class" "A class should capture one, and only one key abstraction with all its information and its entire behavior" "Avoid pure accessor operations"

After this, we have shown how the contents of the MeTHOOD catalogue help the *schema designer* to create such a schema and measure its quality. Table 15 gives an explicit summary of the link between objectbase requirements and the provided measures and heuristics in the MeTHOOD catalogue.

In order to use the contents of the MeTHOOD catalogue for objectbase design, tool support is needed. The following section describes the architecture and prototypical implementation of a design tool for MeTHOOD.

10 MEx: A design support system for MeTHOOD

Using integrated design knowledge efficiently requires tools.

We have observed that it is hard to do a complete "design check" on a complex conceptual design schema "manually". Therefore, we have developed an architecture and a prototype for a *design support system*. This system is called MEx[22]. It is based on a commercial design system and provides automatic support for design monitoring, heuristic checking and fragment transformation, using the design knowledge from the MeTHOOD catalogue.

10.1 MEx Architecture

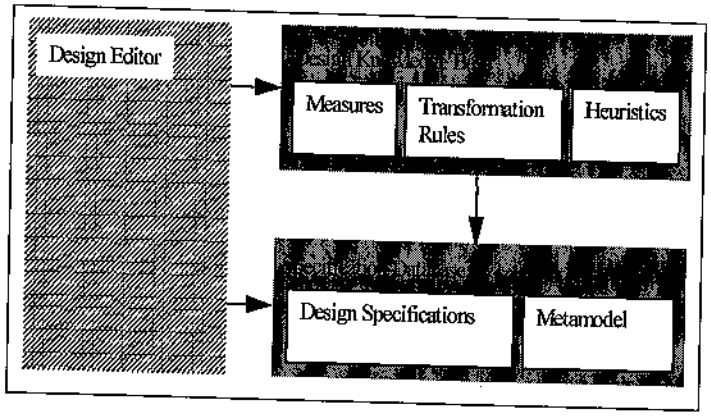


Figure 46: Main components of MEx

The architecture of MEx provides a design knowledge base containing (executable) definitions of measures, heuristics, transformation rules and their relationships. It uses a specification database (with a metamodel of specifi-

[22] MeTHOOD Expert

cation concepts) containing the conceptual design schemas. The main components of MEx are illustrated in Figure 46.

The following sections describe the core features of the architecture of MEx.

10.1.1 Design editor

The design editor provides a graphical view on a conceptual design schema. The primary task of the design editor is to edit the conceptual schema. Furthermore, the design editor provides support for selecting fragments and executing measures, heuristics and transformation rules on these fragments

10.1.2 Specification database and metamodel

The design editor uses a *metamodel* to store the conceptual design schema in the MEx specification objectbase.

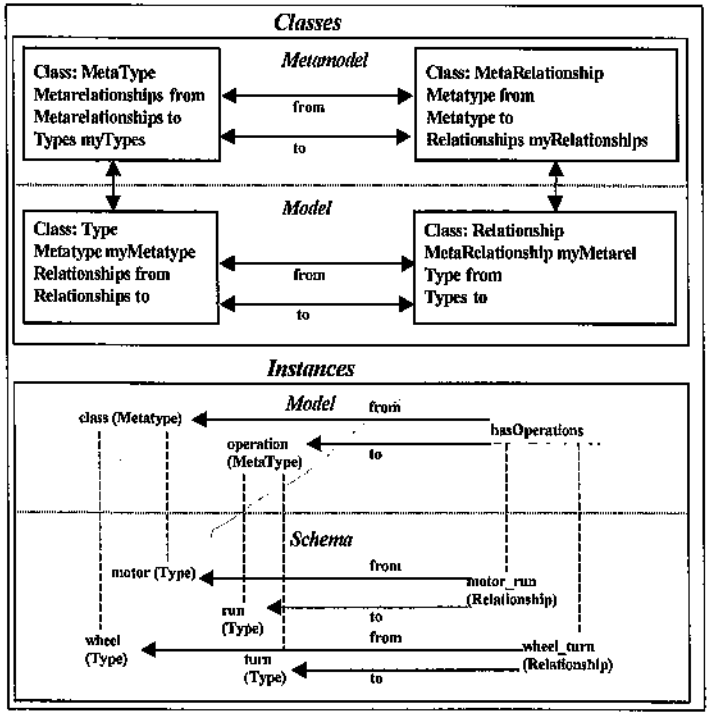


Figure 47: MEx specification database

The metamodel contains information about the allowed modeling constructs (e.g. classes, attributes, operations, ...) and their relationships (e.g. a class

has attributes and operations, a class inherits from other classes). The architecture is based on a dynamic metamodel that allows users to modify the modeling concepts dynamically. This makes it possible to adapt the metamodel to support different object-oriented design methodologies.

Figure 47 shows how the metamodel is related to a conceptual design schema. *Metatype*, *Metarelationship*, *Type* and *Relationship* are the main classes of the specification objectbase. The objects of the classes *Metatype* and *Metarelationship* represent the metamodel. *Metatype* represents the valid modeling concepts; *Metarelationship* represents the relationships between them. In Figure 47, only the modeling concepts *class* and *method* and the relationship *hasOperations* between them are shown as object of *Metatype* and *Metarelationship*, respectively.

The classes *Type* and *Relationship* are the implementation of the model. In the example, their objects *motor* and *wheel* with their operations *run* respectively *turn* from Figure 22 (B) are shown. *motor* and *wheel* are classes in the schema. Therefore, they have the type *class* (denoted by dashed lines), *run* and *turn* have the type *operation*. *run* is an operation of *motor*, *turn* is an operation of *wheel*. This information is represented as objects of the class *Relationship*. In the meta level, *hasOperations* is an object of *MetaRelationship*. This metarelationship associates *class* and *operation* (denoted by arrows). The concrete relationships are captured as the *Relationship* objects *motor_run* and *wheel_turn*.

This representation of a conceptual design schema allows to

- analyze a design specification by means of queries and
- update the conceptual design schema and the metamodel dynamically (at run time).

In fact, the query

```
select * from x in Type where x.myMetatype = "class"
```

delivers the result *motor* and *wheel*, i.e. all elements of type with the metatype *class*. The update

```
insert "wheel" w into Type;
w.myMetatype="class";
```

defines the new class *wheel*.

10.1.3 Design knowledge base

The design knowledge base is the core of MEx. It represents design knowledge in the form of measures, heuristics and transformation rules. It is important to note that measures, checking queries and transformation rules are

stored as queries respectively update specifications. The description of heuristic 11 contains the following checking query:

```
select   x, y, b.from, a
from     x, y in type, a, b in Relationship
where    a.myMetarel. name="sendsMessage"
and      b.myMetarel. name="hasParts"
and      a.from = x and y in a.to
and      x in b.to and y in b.to
```

In MeTHOOD, such a query is applied to a selected design fragment. It delivers a non-empty result if the corresponding heuristic is violated in the fragment. The result is a description of all occurrences of violations of the heuristic in the fragment. The specific format of the result is different for different heuristics. It is used to display violations of the heuristic to the designer and to provide necessary information to transform the violating fragments. If a violation of a heuristic is found, it can be removed using a transformation rule. A corresponding transformation rule to the checking query above is implemented by the update

```
insert b in b.from, y;
remove b from x;
```

which removes the detected violation of the heuristic from the fragment.

Like the information in the specification objectbase, the information in the design knowledge database can be updated. This means that users of MEx are able to modify the design knowledge included in MEx and provide their own measures, heuristics and transformation rules.

10.2 MEx Implementation

We have developed a prototype of MEx to have a tool that helps to automate some of the steps of the MeTHOOD validation (presented in section 12). We selected a commercial design tool (Rational Rose) which already contains a design editor and a specification database with a metamodel. Furthermore, it provides reverse engineering facilities allowing to create design models from C++ and Java source code.

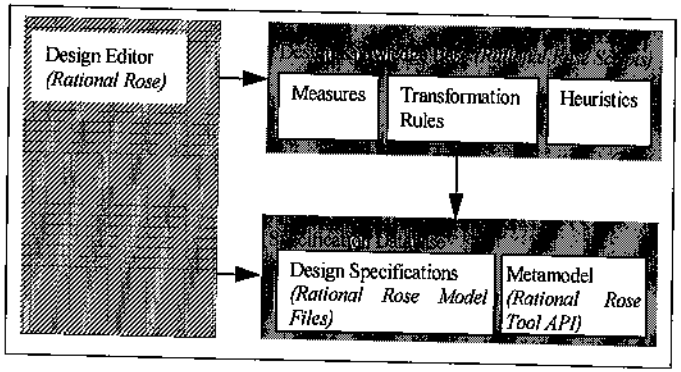


Figure 48: MEx Implementation

The implementation of the MEx Architecture with Rational Rose is illustrated in Figure 48. It consists of a large set of Rational Rose *scripts* (developed using the Rational Rose Tool API). These scripts implement

- all MeTHOOD measures,
- a subset of the MeTHOOD heuristics and
- a subset of the transformation rules.

The implemented subset of the heuristics and transformation rules (including more than 50% of the MeTHOOD catalogue) is determined by their information need. We have implemented those heuristics and transformation rules which do not require

- knowledge of the method source code
- other forms of knowledge (e.g. Functional Dependencies, see 8.9) which is not available in Rational Rose schemas.

The list of implemented heuristics is illustrated in Table 16.

Table 16: Implemented heuristics

Heuristic
A class in a containment hierarchy should only depend on its child classes
Every attribute should be hidden within its class
Avoid dependencies of objectbase classes on their clients
Do not create unnecessary classes to model roles
Avoid additional relationships of base classes to their derived classes
Whenever possible, convert associations and uses relationships in the strongest containment relationship
Common properties of objects should be defined in a single location
Soft classes should not be base classes
The overloading should only define differences to the overloaded operation
Avoid full parallel overloading in siblings

Heuristic
Dependencies in inheritance hierarchies should not go from higher to lower levels
Hard fragments should not depend on soft fragments
Avoid direct recursive associations

Since it would go beyond this thesis to provide a complete set of transformation rules for all heuristics, for every implemented heuristic, a single *default transformation rule* has been implemented. Such a default transformation rule is necessary if a complete schema has to be transformed automatically without any user interaction. This feature was needed for some of the validation executed with commercial projects described in the following part.

Part IV. Validation

11 Comparison to Related Work

We have seen that the ingredients of object-oriented methodologies are mainly notation and some high-level process activities. Apart from the examples in some of the methodologies, none of these approaches supports the designer in getting more experienced and learning from advice. In this sense, current methodologies provide poor support for quality design. This situation led to four different research directions:

1. software patterns
2. heuristic collections
3. design quality approaches and metric suites and
4. database quality approaches

This research had a strong influence on MeTHOOD. In the following part, the commonalties and differences to the most important approaches are examined.

11.1 Software Patterns

The currently available definitions of the term *software pattern* (Gamma, Helm et al. 1994; Coplien, Schmidt 1995; Shaw 1995) (Beck, Johnson 1994; Fowler 1996) (Alexander 1964; Alexander 1979) are:

- A pattern is a [proven][recurring] solution in a context. Each pattern is a three-part rule which expresses a relationship between a certain context, a problem and a solution configuration, which allows to resolve the forces.
- A pattern is a proven resolution of a recurring set of forces within a defined context.
- A pattern relates a system of forces recurring within a context to a proven software configuration that resolves them.

Patterns attempt to capture (and distill!) experience and best praxis from analysts, designers and developers. The MeTHOOD integration schema also

describes a relationship between a context, a problem and a solution. The context is described by heuristics, the problem is described by a set of measures and the solution is described by transformation rules. However, MeTHOOD is different to *design* patterns as described by (Gamma, Helm et al. 1994; Coplien, Schmidt 1995; Coplien, Schmidt 1996).

11.1.1 Design patterns

Design patterns (Gamma, Helm et al. 1994) are software patterns that solve specific design problems. Design patterns are now-well known, and most designers recognize a "real" design pattern. They would not consider a "*measure-heuristic-transformation rule*"-triple (as proposed in the MeTHOOD integration) as a pattern.

We take the point of view that different *families of solutions* for a *family of problems* (described by a single heuristic) can exist. In most cases, this *solution family* cannot be adequately described by a *single* general structure. We describe the solution *implicitly* as a process that marks bad locations in a design and removes recognized misfits. The structure of the concrete transformed design (*the solution structure*) can sometimes (by accident) correspond to the solution structure of a design pattern.

In MeTHOOD, the process of "marking misfit" is supported by explicit, formal descriptions of the context. If we have a checkable heuristic, we can objectively mark every violating part in a design fragment.

The process of finding locations in a design in which a specific design pattern could/should be applied is not supported in current design pattern literature. In general, patterns are better suited to support the "creation" part of design, whereas MeTHOOD supports the "refinement" part better. Patterns are a high level language for building; MeTHOOD provides a high level language for systematic improvement and repair.

Furthermore, design patterns are often focused on very specific structural problems (consider the bridge in (Gamma, Helm et al. 1994), for example). MeTHOOD focuses on more general quality problems (e.g. how can we improve flexibility in this fragment). These quality problems are formally described by measures. The measures are used to accurately define the relationships to forces (force diagrams). Note: It is not possible to describe all forces and all their relationships uniformly and completely, but MeTHOOD attempts to give the best possible view of the diagram of forces according to currently available measures.

The differences above lead to a different purpose of patterns and of the MeTHOOD approach. This suggests that designers should use both approaches in combination. As mentioned above, patterns are better suited to support the "*creation*" and the "*invention*" part of design, whereas MeTHOOD supports the "*refinement*" part better. Patterns are a high-level language for building; MeTHOOD provides a high level language for sys-

tematic repair (and improvement). The relationship between patterns and MeTHOOD can be compared to the relationship between a set of document templates and a tool for document style checking. Designers should use patterns for

- high-level communication about the structure of a design (the *what*),
- documenting the structure of a design,
- sharing design experience in form of design examples,

and MeTHOOD for

- high-level communication about the reasons for design trade-off (the *why*),
- documenting design decisions and
- sharing design experience in form of general rules.

Furthermore, the MeTHOOD approach is better suited for integrating design experience in an automated design support system. The best a system like that could do with a set of design patterns would be "mine" an existing design for occurrences of a pattern. If such an occurrence is found, the designer can "manually" evaluate the design according to problem and context description. MeTHOOD provides measures, heuristics and transformation rules as a design knowledge base for a design support system. The measures continuously monitor design decisions and point out heuristics that should be used in a specific situation. Heuristics are used to automatically check the design for potential design flaws. If such a flaw is found, transformation rules are used to generate proposals for alternative designs. As (Riel 1995) suggests, heuristics can even be used to find locations in which a specific design pattern should be applied.

11.1.2 Analysis patterns

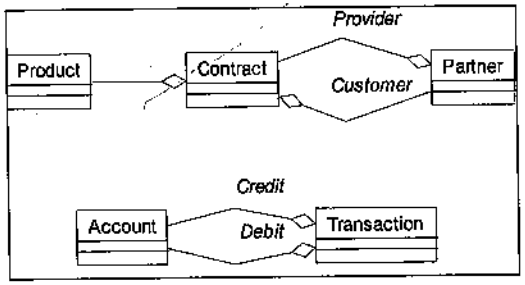


Figure 49: Analysis Patterns

Analysis patterns (Coad, North et al. 1995; Hay 1995; Fowler 1996) are software patterns that solve specific business problems. Examples of such a

business problem are, e.g., the relationships between organizations, their products and their customers, or the way how a financial transaction is handled (see Figure 49: Analysis Patterns).

In most systems we are aware of, a contract is a relationship between products, product providers and customers of these providers. In most of these systems, a transaction relates a deposit and payment on an account. The sum of all actions of a transaction has to be 0.

These and other solutions are needed in most information systems. They provide knowledge about how the business content of a system can be structured. An important work about analysis patterns can be found in (Hay 1995). There, extended entity relation diagrams provide analysis knowledge, especially in the area of manufacturing. (Coad, North et al. 1995; Fowler 1996) provide analysis patterns in an object-oriented notation. (Coad, North et al. 1995) describes experiences made during the construction of several information systems and provides patterns like transaction and product line. (Fowler 1996) mainly provides distilled knowledge out of the area of medical information systems. He also shows how this knowledge can be applied to several other domains.

Analysis patterns are similar to design patterns. The line between analysis patterns and design patterns is hard to draw, especially if the analysis patterns are on a very high level of abstraction. Figure 50 shows how the product-contract-partner pattern from Figure 49 is applied on the business of a travel agency. It shows that the business of the "travel agency" is not visible in the pattern itself.

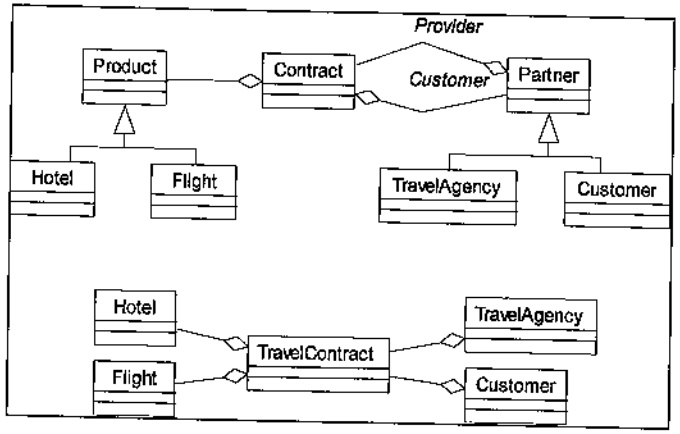


Figure 50: Application of an analysis pattern

Such a high level of abstraction makes analysis patterns more general, however, the business "behind" the pattern is invisible. Another difference be-

tween analysis and design patterns is that the description of most known analysis patterns is on a higher level than the description of current design patterns.

The relation between MeTHOOD and analysis patterns is similar to the relation between MeTHOOD and design patterns. Whereas a pattern is always used to describe a family of solutions, MeTHOOD is used to describe a family of potential problems and the way to a solution.

11.1.3 Information Integrity Patterns

The *Checks Language* (Cunningham 1995) is a pattern language that contains different patterns with advice for the design of a user interface and a *domain model* (which is another term for objectbase schema). These patterns reduce the complexity and the effort of constructing the relationship between user interface and domain model. They give advice how to support checks on user inputs that could violate the integrity of an objectbase. The most important pattern in this language is *Whole Value*. It states that some special, constructed classes (e.g. currency, period and telephone number) should be used to represent quantities in the domain, instead of atomic values like number, strings and others. These classes should

- be independent from the particular domain and
- include format converters and
- allow only "correct" values.

In many systems, these classes are used to provide very early checks of values (which can occur even before the value is assigned to an object. Other patterns in this language include *Exceptional Value* (how to handle valid user input outside of Whole Values), *Deferred Validation* (when to check a complete series of inputs) and *Visible Implication* (how to handle values which can be derived from others).

11.2 Antipatterns

Andrew Koenig has coined the term *antipattern* in (Koenig 1996). There is no exact definition for the term antipattern. Koenig defines an *antipattern* as a pattern, except that "instead of a solution, it gives something that looks superficially like a solution, but isn't one". So, one could define an antipattern as: "An antipattern is a [proven][recurring] *wrong* solution in a context". The description of the Big Ball of Mud, as a casually haphazardly structured system in (Foote, Yoder 1997) is an example of this. Koenig and others suggest that pattern - antipattern pairs could be used to transform a bad design into a good one.

The fundamental problem of this approach is that "For every way of doing something right, there are a dozen ways of doing it wrong"[23]. However, antipatterns can be seen as an extension of patterns, as they represent traps and pitfalls concerning the patterns. They can also be seen as a learning tool that helps people to learn from other people's mistakes and to recognize early on where one starts to go wrong.

Heuristics are loosely related to antipatterns; both describe some forms of misfit. However, antipatterns describe the specific form of misfit that can occur if the problem solved by the corresponding pattern is solved the wrong way. In our opinion, such specific problem descriptions are too numerous, very hard to specify and useful only in the context of an existing pattern (to avoid modifying and applying the pattern is a clumsy way).

11.3 Heuristic collections

Both Rumbaugh and Henderson-Sellers agree that heuristics are a necessary ingredient in a good software methodology (Henderson-Sellers 1995; Rumbaugh 1995). Most current methodologies do not consider them explicitly. In some cases, they appear implicitly in the description of a methodology. However, recent publications (Riel 1996) (Booch 1995; Coad, North et al. 1995) explicitly include heuristics and patterns. They focus on project management, organizational issues, the development process and other issues. A few heuristics address the general design of classes, e.g., "Every class should embody only about 3-5 distinct functionalities".

11.3.1 The Riel heuristics

The most important collection of design heuristics is contained in (Riel 1996). This collection is categorized into:

- general heuristics for classes and objects,
- heuristics which prevent action-oriented (or procedural) applications,
- heuristics considering different kinds of relationships between classes and between objects, including collaboration, containment, inheritance, association, multiple inheritance and
- heuristics for physical object-oriented design.

(Riel 1996) also describes a relationship between heuristics and design patterns. Riel states that heuristics can be used to identify new design patterns in a given design (as a tool for pattern mining). He also introduces the notion of *transformation patterns*. A transformation pattern is a pattern that captures the operation by which a bad design is transformed into a good design. The examples in (Riel 1996) suggest that a transformation pattern can be considered as a pattern pair consisting of a "good" and a "bad" pat-

[23] Comment by Jim Coplien in a newsgroup discussion.

tern which are "linked" by a heuristic. The heuristic states why the bad pattern is bad and motivates the good pattern. The bad pattern violates the heuristic, the good pattern does not.

The notion of heuristics is identical in (Riel 1996) and in MeTHOOD, and transformation patterns in (Riel 1996) are similar to the transformation rules in MeTHOOD. The main differences between (Riel 1996) and MeTHOOD are twofold:

MeTHOOD does not include an explicit description of the bad and the good pattern in a transformation rule. A single heuristic in MeTHOOD describes a family of design flaws. It is not always possible to describe this family by a single structural pattern. However, the checking rule is able to recognize all objects included. Therefore, in MeTHOOD, the "bad pattern" of a transformation rule would be redundant. The same is true for the good pattern part of a transformation rule. An identified design flaw can be removed in various ways.

These ways are often even influenced by the current development phase (e.g., conceptual design and physical design) or desired quality properties of

the design. In MeTHOOD, these ways are described by transformation rules associated to heuristics. Such a rule is a description of a real process, e.g., 1. "remove this association", 2. "insert an inheritance relationship here".

This process can result in many different shapes of the solution's structure. This family of shapes cannot always be described by a single "good pattern". This means that the conflict resolution concepts of MeTHOOD are simpler, because, besides heuristics, only transformation rules describing a process are needed. Moreover, they are more flexible, because many different problem patterns can be recognized, and many different solution patterns can be generated.

(Riel 1996) recognizes that different heuristics can collide, which means, they give different conflicting design advice for the same design fragment. (Riel 1996) suggests that the designer has to resolve these conflicts and proposes only limited support for this kind of decisions. MeTHOOD uses measures to help during conflict resolution. In MeTHOOD, the designer can look at the measures for the resulting alternative design fragments. This helps him to choose the best one. Furthermore, in MeTHOOD, heuristics are weighted according to their importance during different phases of the development cycle. If the weighting is different for conflicting heuristics in a given development phase, this provides further selection criteria for the designer.

11.3.2 The Martin principles

(Martin 1995a) includes a collection of nine design "principles":

1. The *open/closed* principle: "Software entities should be open for extension, but closed for modification." (Design ideal)
2. The *Liskov Substitution* principle: "Derived classes must be useable through the base class interface without the need for the user to know the difference." (Checkable design heuristic)
3. The *dependency inversion* principle: "Details should depend upon abstractions. Abstractions should not depend on details." (Design ideal)
4. The *granule of reuse is the same as the granule of release*. (Organizational rule)
5. Classes within a released component should share *common closure*. (What effects one, effects all.) (Organizational Rule)
6. Classes within a released component should be reused together. It is impossible to separate them. (Organizational Rule)
7. There are no cycles in the dependency structure of released components. (Checkable design heuristic)
8. Never cause a category to depend upon less stable categories[24]. (Checkable design heuristic)
9. a) The more stable a class category is, the more it must consist of abstract classes. (Design ideal) b) A completely stable category should consist of nothing but abstract classes. (Checkable design heuristic)

These "principles" include design ideals (1., 3., 9a), organizational rules (4., 5., 6.) and checkable heuristics (2., 7., 8., 9b). Unlike MeTHOOD, this collection does not include the notion of transformation rules. Some of the heuristics are supported by measures (8. , 9.).

11.3.3 TOAD

(Gibbon, Higgins 1996) describes a prototypical tool which is called TOAD (Teaching Object-Oriented Analysis and Design). This tool is able to examine static class properties captured in C++-header files based on a set of heuristics. The heuristics are similar to the heuristics in the MeTHOOD catalogue. TOAD does not contain transformation rules or measures.

11.4 Refactoring

Refactoring is a term describing design techniques that make transformation of a program structure easier for a designer (Opdyke 1992). (Johnson, Opdyke 1993) notices that programs are often refactored to make them more reusable or easier to maintain.

[24] The heuristic has been changed from the original because it is closer to the core of the statement.

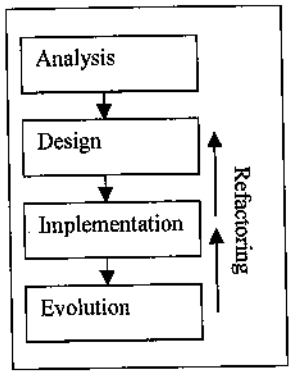


Figure 51: Refactoring

Refactoring a program means keeping existing functionality and behavior. The most complete work about refactoring is (Opdyke 1992). Refactoring techniques are used on the code level, currently mainly on Smalltalk programs. They are applied when the structure of the available source code gets bad or when its general design has to be changed. (Johnson, Opdyke 1993) have developed a *refactory* for Smalltalk. A refactory is a program that supports developers during refactoring. A refactory is always programming language dependent since it has to analyze the program. A refactoring is a description of a process that executes a specific program transformation. An example of refactoring is changing the name of a variable or a class. (Johnson, Opdyke 1993) defines refactoring for high level transformations as moving variables or functions to components and moving members to aggregates.

The techniques described in MeTHOOD and refactoring complement each other. MeTHOOD proposes different transformation rules that transform a design with a flaw into a better design. In our validation (see chapter 12), we have obtained the design specification from the source code of an existing system. MeTHOOD proposed various *design* transformations that we wanted to propagate in the source code. A refactory able to execute these transformations on the source code would have been extremely helpful. In this sense, both approaches provide and automate some kind of design knowledge. On the other hand, they are different, because MeTHOOD helps designers to make design decisions and provides transformations on the schema level, whereas refactoring provides knowledge about subsequent changes in a program, hence helping designers to execute design decisions. If a developer decides to make a change in a program – the refactoring knowledge helps him to execute all subsequent changes without introducing new bugs.

11.5 Design Quality Approaches and Metric Suites

Current approaches in quality modeling examine what software quality means, how it can be measured and predicted. A good quality model can be used to monitor the quality of the design and give the designer feedback for his design decisions.

A large amount of approaches has been proposed in the areas of object-oriented metrics and quality models including (Chidamber, Kemerer 1994) (Abreu, Carapuca 1994) (Lorenz, Kidd 1994) (Abbott, Korson et al. 1994) (Frakes, Terry 1996) and (Boehm 1978) (Harrison 1993) (Fenton, Whitty et al. 1995) (Kitchenham, Pfleeger et al. 1996) (IEEE 1989) (ISO 1990). Most of them use measures and relationship models between measures (quality models) to define what software quality is. The main difference between these approaches and MeTHOOD is the inclusion of heuristics (which are not implied by measures) and transformation rules. MeTHOOD not only includes a definition of quality and the way it can be measured (based upon the above approaches), it also explicitly describes how to improve it. There are some other quality approaches providing this kind of more "active" design support (see sections 11.5.2, 11.5.3, 11.5.4).

11.5.1 The Chidamber and Abreu metrics

The metrics suite (Chidamber, Kemerer 1994) is a well-known collection of metrics which is supported by different tools. The metrics are:

Weighted methods[25] per class (WMC): sum of complexity of all methods in the class
Depth of inheritance tree (DIT): how deep is a class in the inheritance hierarchy
Number of children (NOC): number of immediate subclasses of a class
Coupling between object classes (CBO): count of classes coupled with the given class
Response for a class (RFC): number of methods, which could be executed after message is received by the class
Lack of Cohesion in Methods (LCOM): null intersections - number of nonempty intersections between sets of used attributes per method

The metrics suite proposed in (Abreu, Carapuca 1994) contains the following metrics:

[25] "Method" in these definitions is a synonym to "operation" in the MeTHOOD terminology.

Method hiding factor (MHF): number of all hidden methods/number of all methods
Attribute hiding factor (AHF) : number of all hidden attributes/number of all attributes
Method inheritance factor (MIF): number of inherited (not overridden) methods/number of all methods (new, inherited, overridden)
Attribute inheritance factor (AIF): number of inherited (not overridden) attributes/number of all methods (new, inherited, overridden)
Coupling factor (COF): number of all disjunctive pairs of classes with a client-server relationship/number of disjunctive pairs of classes
Clustering factor (CLF): Number of disjoint graphs made up of classes and their relationships/number of classes
Polymorphism factor (PF): maximum number of possible different polymorph situations (all operations overridden)/number of possible different polymorph situations
Reuse factor (RF): considers library reuse + inheritance reuse

Both metric suites have a strong influence on the definition of the MeTHOOD measures. The fundamental difference between the metrics proposed here and the MeTHOOD measure is that the MeTHOOD measures are more sensitive because they take the size of attributes and operations into account. The MeTHOOD measures "recognize" type changes and additional operation parameters that are not recognized by simple attribute and operation counts. The MeTHOOD encapsulation measure is an extended combination of MHF and AHF and WMC. The MeTHOOD coupling measures are similar to COF. Some of the proposed metrics are used to recognize violations of a heuristic, e.g., DIT for the heuristic "Dependencies in inheritance hierarchies should not go from higher to lower levels" Variations of MIF and AIF are used within the MeTHOOD size model. And LCOM inspired the MeTHOOD cohesion measure, which later turned out to get a complete different definition. NOC, RFC, CLF, PF and RF are not considered in MeTHOOD. Partly because they are hard to collect based on the information we assume, and partly because we do not see how they can be used to improve a design.

11.5.2 The Law of Demeter

(Lieberherr, Holland 1989) describes a rule for object-oriented programming which is known as the "Law of Demeter". This "law" is a restriction on the set of possible message receiver objects in a method. It is a typical example of a very powerful heuristic. The "object form" of this rule can be formulated as: "An operation should send messages only to objects it has directly created, the current object (self, this), its components, objects in its parameterlist or in global variables ". Like other heuristics, the "Law of Demeter" is considered as a guideline for good design, and not as an absolute restriction. Associated to the rule is a technique that allows to transform designs which do not comply to the law. This is a typical example of a transformation rule.

11.5.3 The Martin Metrics

(Martin 1995) describes a set of important metrics which are used to measure the quality of an object-oriented design. The key idea is to measure the stability of class categories and to reduce dependencies from non-stable categories. The stability of a category is determined by forces (in (Martin 1995) dependencies from another category) which could lead to changes in it. Not all categories should be as stable as possible (this would make the system hard to change). The key to making categories flexible are abstract classes because they can be extended without requiring subsequent changes. Stable categories, which have built-in extensibility by abstract classes, are desirable. Therefore, (Martin 1995) relates stability and abstractness and defines a further metric, "Distance from the main sequence", which allows to recognize classes which are not too abstract for their stability and not instable in spite of their abstractness. This metric implies the design heuristic: "Class categories should be close to the main sequence."

11.5.4 Pattern-Lint

(Sefika, Sane et al. 1996) has implemented a tool which integrates static analysis and dynamic visualization and checks for a variety of design principles like coding styles, architectural rules like design patterns or styles and heuristic guidelines like coupling and cohesion.

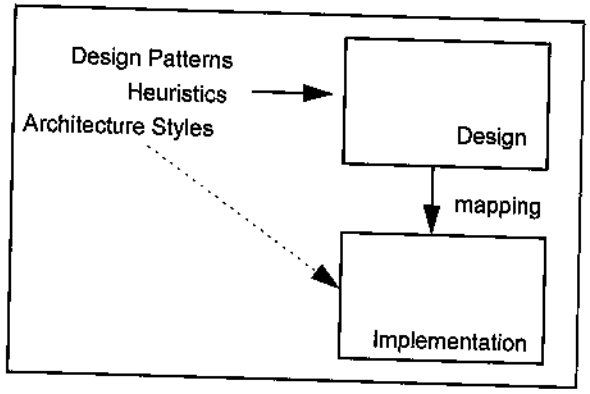


Figure 52: Pattern-Lint

The tool is used to verify that good design principles which have been used for design are not obscured in the implementation, e.g., the principle that a mediator pattern in design is a mediator pattern in implementation. Furthermore, it allows checking of coupling and cohesion at code level. The main benefit of this approach is the integration of static structure checking and dynamic visualization, which allows run-time checking of the behavior of a subsystem. The approach distinguishes between low-level rules (closer to implementation than to design), which are specified in ER diagrams, and

architectural rules (patterns, software interconnection models, architectural styles), which are specified as prolog clauses.

The main difference between MeTHOOD and Pattern-Lint is twofold:

1. MeTHOOD does not check whether design rules which are present in conceptual design are preserved during implementation; MeTHOOD assures that design rules are not violated in conceptual design.
2. MeTHOOD extends the pure checking and visualization capabilities of Pattern-Lint because it includes also generative concepts, which are used to provide advice if a violation is detected.

11.5.5 GRCM model

(Tervonen 1996) has extended the SQM (Software Quality Metrics) (Boehm 1978) model to a quality model which includes design rules. His model is called GRCM (Goal Rule Checklist Metric) model. GRCM provides assistance for developers and design inspectors. According to a checklist, they can evaluate the quality of a design and improve it.

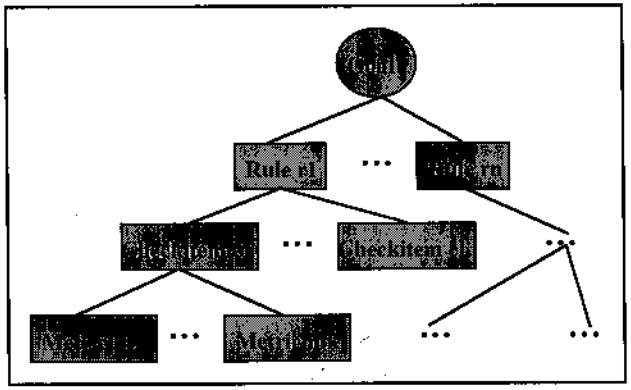


Figure 53: GRCM model

GRCM includes a supporting tool set for training design. The focus of GRMC is on training and design inspection. Checklists support design inspection. The items in a checklist in GRCM correspond to implied heuristics in MeTHOOD. GRMC does not include design heuristics that are associated to metrics. GRMC does not support a prioritization of checklist items. Furthermore, GRMC is more focused on organizational issues of quality design than MeTHOOD, e.g., the process also includes inspections by "external" inspectors. MeTHOOD focuses on continuous inspection by the developer. Besides the inspection aspect, MeTHOOD also considers design improvement, using transformation rules. This aspect is not considered in GRMC.

11.6 Database quality approaches

11.6.1 Quality criteria for conceptual database schemas

Rishe (Rishe 1988) attempts to support designers achieving high quality schemas by providing a special modeling language (the Semantic Binary Model) and claims that it is impossible to describe a miniworld by a schema in other models (relational model, network or hierarchical model) with the same quality.

(Rishe 1988) gives some necessary criteria which have to be fulfilled by a high-quality schema:

1. It describes the miniworld naturally: It describes the objects as they are in the real world. Users can easily translate ideas in both directions.
2. The schema contains very little or no redundancy: Redundancy is the possibility that a fact is repeated. Redundancy should be avoided to prevent inconsistency.
3. The schema does not impose implementation restrictions. Every situation probable in the real world is fully representable under the schema.
4. The schema by itself covers as many integrity constraints as possible.
5. The schema is flexible. Small changes should not drastically propagate.
6. The schema is conceptually minimal. It does not involve concepts that are irrelevant in the application's miniworld.

The MeTHOOD approach is different. It does not provide a modeling language, but a set of guidelines. These guidelines also consider some of the quality criteria required by Rishe (although some of the criteria are slightly differently defined).

I. *"Natural" description of miniworld*: This criterion is hard to achieve. It should be considered that different users have a different view of the miniworld. MeTHOOD support for this criterion is described in section 9.2.
2. *Minimized redundancy*: MeTHOOD has a similar notation of redundancy. Support for this design task is considered in section 9.4.
3. *Implementation restrictions*: This criterion is not considered in MeTHOOD. We would argue that the requirement above is impossible to fulfill.
4. *Integrity constraints*: MeTHOOD considers only schema internal integrity constraints. These are considered in section 9.3.
5. *Flexibility*: MeTHOOD provides different measures and heuristics to improve flexibility. These are considered in section 9.5.
6. *Minimality*: This criterion is not considered in MeTHOOD. The heuristics and measures in MeTHOOD cannot assure if some elements of the schema are relevant in the miniworld or not.

Besides these criteria, MeTHOOD also considers some fundamental characteristics of persistent objects which represent the mini-world in section 9.1 and the relationships between the applications and the database schema in section 9.6.

11.6.2 Information modeling

(Kilov, Ross 1994) proposes a disciplined approach to object-oriented analysis with a strong focus on modeling real world entities. They propose a contract-based approach that allows specifying classes as contracts. A contract is the definition of an operation. Besides the description of a signature, a contract also specifies the semantics of an operation. The semantics of an operation are described using invariants that must hold before and after an operation. These conditions are similar to pre- and post-conditions in Eiffel (Meyer 1988).

The approach in (Kilov, Ross 1994) is different to other approaches for extended entity relationship modeling (Lazimi 1989; Navathe 1989; Ku 1991; Nachouki, Chastang 1991; Spaccapietra, Parent 1992), because it puts a strong emphasis on the *fundamental* techniques for object-oriented modeling, namely abstraction and information hiding. Their approach allows specifying information models formally using information hiding and abstraction. Furthermore, they elaborate an association concept based on invariants. An association is defined as a set of objects with an invariant that must hold. This allows them to formally integrate associations and operations: the invariants of the associations must hold before and after each operation. Besides this, Kilov and Ross propose a set of *guidelines for information modeling*:

- Choose the right level of abstraction.
- Ask questions.
- Think in terms of contracts.
- Treat cardinality as a less important detail.
- Question symmetric relationships: expect asymmetry.
- Split parts of recursive associations.
- Determine the amount of flattening.
- Show ties (contracts) to other applications.
- Do not use "code" domains.
- Do not treat like properties as entities: use domains.
- Baseclass associations.

These guidelines include organizational rules (2., 3., 4., 8., 9.), uncheckable heuristics (1., 7., 10., 11.) and checkable heuristics (5., 6., 8.). These guidelines are not supported by measures. A notion of transformation rules does not exist.

11.6.3 ER design with rules and heuristics

(Batra, Antony 1994) describes an experiment with novice designers which shows that these designers make systematic errors when designing a complex schema. The main problem is described as the lack of feedback in conceptual design. (Batra, Antony 1994) claims that tools are needed which provide feedback and propose alternatives for different representations. The later approach (Batra, Zanakis 1994) describes heuristics as simple procedures that are used to easily and quickly provide good, but not necessarily optimal solutions to difficult problems. They distinguish between heuristics and rules that can be formally derived under certain assumptions. Seven rules and three heuristics are presented that should be followed in determining the relationships between entities. The rules focus on the sequence in which different kinds of relationships (one-one, one-many, ternary one-one) are identified. From these rules, a step-by-step design process is given. The rules are not integrated with measures. The schema transformation aspect is considered only for the given examples.

11.6.4 IDEA

IDEA (Ceri, Fraternali 1997) is probably the most extensive database design methodology that takes advantage of object-oriented concepts. It puts a strong emphasis on data modeling and attempts to close the gap between the concepts provided by current object-oriented design methodologies and the concepts needed to design a data intensive application. IDEA is an outcome of a four-year Esprit project and has been developed by different European software organizations and four research institutions (ECRC, INRIA, University of Bonn and Politecnico di Milano).

IDEA proposes a design process consisting of analysis, design, prototyping and implementation. The main contributions of IDEA are techniques for the design of objects and rules during design, prototyping and implementation. The core idea of IDEA is to transfer semantic knowledge from applications to the database. This principle is called "knowledge independence", and the authors claim that knowledge independence simplifies application design, maintenance and evolution. The required "knowledge specification" of the database level is facilitated with a new specification language called Chimera, developed by Berttino, Ceri and Manthey. A suite of design tools supports IDEA. These include

- a graphical object model editor/translator (Iade)
- an active rule analyzer (Arachne)
- a mapping tool for Oracle (Pandora)
- a prototyping tool with update propagation for the management of derived data,
- a schema evolution tool and
- a passive rule design tool (user guidance for deductive rules and detection of inconsistencies)

The proposed object model is a traditional extended entity relationship model which does not allow the specification of operations. To specify the dynamics of the system, traditional state charts (Harel 1987) are proposed. Furthermore, a conceptual language for schema design and the design of deductive and active rules, operations and constraints is proposed.

IDEA allows specifying integrity conditions during analysis. The analysis results are the input for design. During design, different design decisions are required, for example, if an attribute type has to be modeled as a type without object identifiers or a class. Furthermore, different schema optimizations are proposed. These schema optimizations consider potential flaws in the schema and propose a transformation for this flaw. Examples of such a potential problem are classes with a large number of attributes and operations. Such a class should be split into two classes (with different subsets of attributes that are never accessed together) and include a relationship between the two classes. These kinds of operations can be compared to very basic heuristic and transformation rules.

IDEA supports consistency checks of the schema which avoid issues like acyclic generalization relationships. These consistency rules can be compared to the ISA1-ISA4 properties in (Hörner, Heuer 1991).

The core feature of IDEA is the integration of active and deductive rules into the object model. In this extended object model, it is possible to assign preconditions to operations of classes and to define class based integrity constraints.

11.7 Discussion

Like the other approaches presented in section 11, MeTHOOD provides knowledge based on concrete experiences made by designers. However, most of the other approaches consider only a single view of the design knowledge representation, e.g., a set of measures, a set of heuristics or a pattern language. This results in various problems:

- measure sets do not provide concrete advice about *how* to design, they are general indicators,
- heuristics provide concrete advice "how to do", however, they can have conflicts between each other, and there are no defined criteria that help to resolve the conflict,
- heuristics only mark locations, but they do provide advice on how to improve these locations.

Design patterns provide more concrete advice on how to solve concrete design problems in a specific context. However, design patterns are often very specific – and they provide only limited help evaluating the effects of a pattern on a design – furthermore, they do not include a systematic way to indicate locations *in which* a pattern should be applied.

The main drawback of MeTHOOD, compared to the other approaches, is that it is more complicated. To use MeTHOOD efficiently, tool support is required (most patterns and heuristic approaches do not require tool support).

However, MeTHOOD includes three different views on the provided design knowledge. These views are described in the MeTHOOD integration. The heuristics allow detecting locations with potential design flaws. They also can provide conflicting advice, however, the proposed measures can be used to evaluate the effects of the conflicting heuristics more objectively. This solves the major problem of pure heuristic approaches. From another point of view, measures can be used to define quality – and the measured values can indicate how good a schema is. The relation between measures and heuristics allows a systematic improvement of measured values. This solves a major problem criticized in pure measure approaches – the passiveness of measures. Heuristics are only one single step towards a more "active" design support. Another problem of pure heuristic approaches is that they indicate the location *in which* a design could and should be improved; however, they do not indicate *how*. Thus, pure heuristics are passive, too. MeTHOOD proposes more active support by relating heuristics and transformation rules. These patterns transform the structure of a design – and the effects of this restructuring can be measured. Although the resulting structures are similar, a transformation rule can create a family of structures (depending on the source schema). Therefore, transformation rules are more general than patterns, and we suggest using both transformation rules and patterns. Often, designers attempt to improve a design through well-known design patterns. Many transformation rules propose transformations that are instantiations of design patterns. Therefore, heuristics can be used to detect these places in which some of them should be applied.

To support these claims and to evaluate MeTHOOD in a realistic context, we have performed a validation with some industrial applications.

12 Application to existing Java and C++ Systems

To examine if MeTHOOD can be usefully applied to realistic applications, an in-depth evaluation has been executed. For this evaluation, *six* different projects have been selected, and MeTHOOD measures, heuristics and transformation rules have been applied to 2203 classes in 87 components. The following sections illustrate the results of this evaluation. It would go beyond the scope of this document to present all measured values (about 27500) and all the obtained heuristic conflicts and transformations (about 11600). Therefore, only *component measures* (based on packages) and a *selected set of heuristic violations* are presented[26]. The next sections describe *the process* used for the evaluation, *a template* used to capture the evaluation results in a structured manner, the *six project evaluations* and an *overall conclusion.*

12.1 Evaluation process

The used evaluation process is outlined in Figure 54. We have obtained the source code of all examined projects. A set of important values (the MeTHOOD measured properties for *all packages* in every project) has been *estimated* by the authors of the source code. For this estimation, those Projects have been selected in which the project has mainly been developed by a *single author* who was available for an interview. The authors have obtained the names and textual definitions of the mesures, but not the formulas. The result of this estimation are the estimated original properties as described in 12.2.1.2. Please note that these estimations are based on the source code, and not on the reverse engineered schema.

[26] All other measurements, heuristic conflicts and the publicly available schemas (TOS, iBus and JDK 1.1.5) (original and transformed) can be obtained from the author. The complete results are available as follows: the original and transformed models are available as *Rose 98 (Version 4.5.8054a)* model (.mdl) files, all measurements (class and category measurements) are available as excel sheets and all heuristic validations and transformations are available as excel sheets.

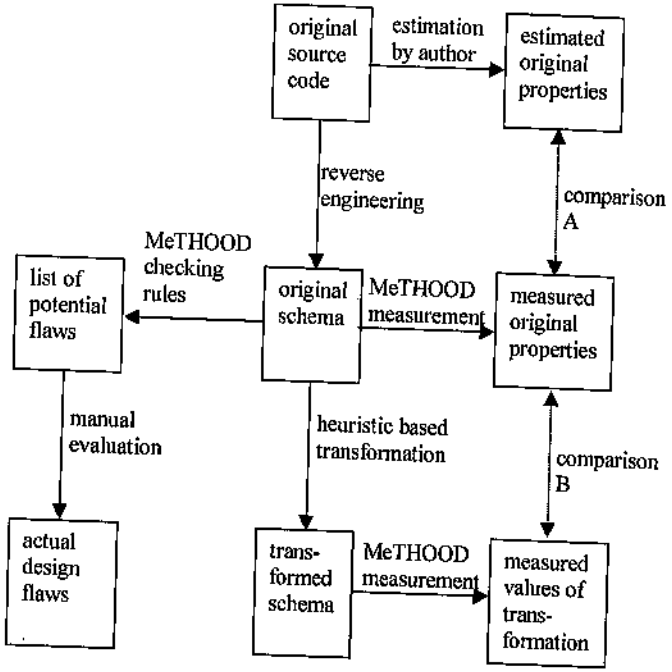


Figure 54: Setup of the evaluation

The source code has been *reverse engineered* using the reverse engineering facilities of the used design tool. Using MeTHOOD, the corresponding properties are *measured* based on the reverse engineered schema. The result of this measurement is described in "12.2.1.3 Measurement results of the original project".
The obtained values of the estimated and the measured properties are *compared* (comparison A). Please note that the measurement is performed on the reverse engineered schema that does not contain information about the implementation of methods. Therefore, the result of the comparison can be used to make statements about

- how well the measures are named and how accurate model properties are measured and
- how MeTHOOD measures can be used to predict properties of implementation, using a schema of the implementation.

The results of the comparison are described in "12.2.2.1 Evaluation of Measures: Comparison between estimated properties and measured results".

Using the MeTHOOD checking rules, a list of potential design flaws is obtained for every project. This list is manually evaluated. The evaluation is described in "12.2.2.2 Evaluation of heuristics: Specific comments to list of heuristic violations".

Furthermore, the original schema is transformed, using checking and transformation rules. The transformed schema is *measured*, using the MeTHOOD measures. The measured properties of the transformed project are described in "12.2.1.5 Measurement results of transformed project". The measured values of the transformed project are *compared* to the measured properties of the original project. The result of this comparison is described in "12.2.2.3 Evaluation of heuristic based transformation: Comparison of measures between before and after the transformation". The summary of all comparisons and evaluations of all projects is described in 12.8.

12.1.1 Mapping of heuristic identifiers

MEX uses heuristic identifiers to distinguish between the different heuristics. Table 17 includes a mapping between the identifiers of the implemented heuristics and the heuristics described in the MeTHOOD heuristics catalogue.

Table 17: Mapping of heuristic identifiers

Heuristic identifiers	Heuristic
MH02	8.3 Every attribute should be hidden within its class
MH06	8.6 Do not create unnecessary classes to model roles
MH08	8.8 Avoid additional relationships of base classes to their derived classes
MH11	8.11 Whenever possible, convert associations and uses relationships in the strongest containment relationship
MH14	8.14 Common properties of objects should be defined in a single location
MH15	8.15 Soft classes should not be base classes
MH17	8.17 The overloading should only define differences to the overloaded operation
MH18	8.18 Avoid full parallel overloading in siblings
MH23	8.23 Dependencies in inheritance hierarchies should not go from higher to lower levels
MH24	8.24 Hard fragments should not depend on soft fragments
MH25	8.25 Avoid direct recursive associations

12.1.2 Assumptions and restrictions of the evaluation

Currently the MeX prototype has the following restrictions:

1. The method of comparing estimated properties of source code with measured properties of a reverse engineered schema has the disadvantage that estimations and measured results will vary if important properties are "buried" in the method source code. This is the case for "closed" methods which perform many tasks on many objects in the schema, using only local variables. Typical examples for this type of methods are *example* and *test* classes where a *main method* performs some *test* and *examples*, using many classes of the schema. The estimated results of this type of classes will vary from the measured results, because the source code is not visible for the measures. However, in the examined projects, the classes in which these variations occur do not play the role of a server or objectbase class – therefore, these variations are not important for this evaluation.
2. Due to a limitation of the metamodel API (application programming interface) of the used design tool, dependencies based on Java exception classes are not correctly recognized.

12.2 Template for describing MeTHOOD evaluation based on a specific Project

This template has two objectives:

1) To describe the *measurement estimations* and the results of the *project assessment:* The measurement estimations are estimations of properties which are formulated by project members and me. They are used to validate whether the measured results are close to the estimations. These estimations are used to perform an empiric evaluation of the measures. The project assessment describes a subset of facts observed and measured in the original project. It is used as a basis for a comparison with the transformed project.
2) To describe the *results of the MeTHOOD evaluation* from the projects' point of view: The result of the evaluation describes the differences between the measurement estimations and the measured results. These differences are used to validate the measures. Furthermore, specific locations in the original project which violate heuristics are evaluated. The evaluation is a subjective decision whether the heuristic has been successful and the location really should be improved or whether the heuristic has failed and found a location which cannot be improved. Finally, the measured results of the original project are compared with those of the transformed project. Assuming that the measure evaluation is positive, the difference can be used to judge whether the properties of the fragments have been improved by the heuristic based transformation.

The template, therefore, consists of two sections: the *project assessment* and the *empiric evaluation*.

12.2.1 Project Assessment

12.2.1.1 Description of the project architecture

In this section, we give a short description of the high-level architecture and the general idea of the project. There is also described how the project is subdivided into packages.

12.2.1.2 Estimated properties of the original project

In this section, estimated properties of different interesting fragments are illustrated. Some of the properties are derived from the role of the component in the project, others have been collected in interviews with the developers.

12.2.1.3 Measurement results of the original project

In this section, the results from the MeTHOOD measurement are presented. It contains a section with all measured values for all classes, a section in which fragments are components and a last section in which the project is the fragment. Furthermore, different derived measures are presented (e.g. average size of class).

12.2.1.4 Heuristic violations in original project

This section contains a description of some important *heuristic violations* of the project. A heuristic violation is a location in the original project in which a MeTHOOD checking rule has detected a conflict to a heuristic.

12.2.1.5 Measurement results of transformed project

This section has the same structure as section 12.2.1.3. It contains the same measurements on the transformed project.

12.2.2 Empiric Evaluation

12.2.2.1 Evaluation of Measures: Comparison between estimated properties and measured results

The objective of this section is to validate the applied measures. The results from sections 12.3.1.2 and 12.2.1.3 are compared. The measured results should be as close as possible to the estimations.

12.2.2.2 Evaluation of heuristics: Specific comments to list of heuristic violations

This section comments the heuristic validations from section 12.3.1.3. For selected interesting violations a statement is made whether the heuristic violation is a design flaw or not.

12.2.2.3 Evaluation of heuristic based transformation: Comparison of measures between before and after the transformation

This section compares the measured results from sections 12.2.1.3 and 12.2.1.5. If the evaluation of the measures in 12.2.2.1 is positive, this section evaluates the complete MeTHOOD approach and therefore contains the results of the project evaluation.

12.3 TOS (Traveling Object System)

12.3.1 Project Assessment

TOS is an infrastructure for mobile Java agents. The main idea of TOS is to create a bundle consisting of instance data (attribute values of a Java object) and the class of the object. This bundle is called *TravellingObject* and can be transferred to another node in the network in order to be executed. TOS has been developed at the Systor research lab. The present examination is performed using the TOS class library, a library of Java classes which can be used to build TOS based applications.

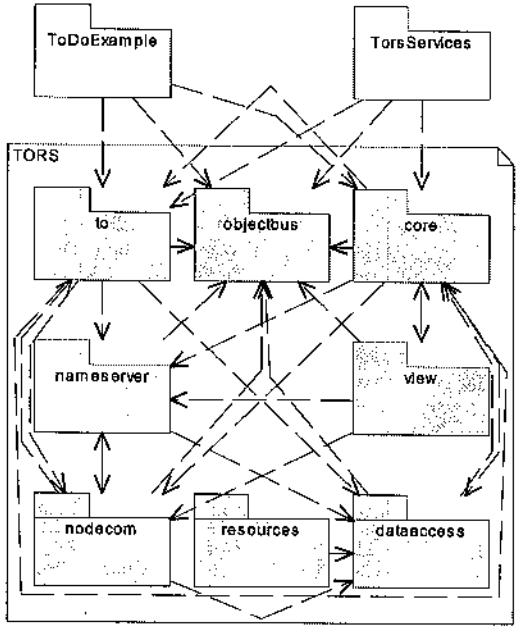


Figure 55: Architecture of TOS

12.3.1.1 Description of the project architecture

Figure 55 shows an overview of the architecture of TOS. It mainly consists of two parts: the traveling object runtime system (TORS) and the traveling objects (TOs). The TO implementations are bundled into the packages TorsServices and ToDoExamples and can be considered as the clients of the TORS.

TOs can perform different tasks if they are executed. Implemented examples for these tasks are a distributed ToDo-list and a help desk application that allows to create problem tickets and route them through a network of help desk employees.

TORS is their execution infrastructure. A TORS has to be installed on every node at which TOs are executed. The TORS runs in a Java process, listens to a specific port, receives TOs and executes them. As shown in the figure, the TORS contains different components:

TorsServices	Specific traveling objects which perform different tasks
tors.view	The UserInterface of the TORS. Used to manage nodes in the network, and manage and send TOs
tors.to	An abstraction of the TO.
tors.resources	Language specific resource bundles for the TORS.
tors.objectbus	A publish/subscribe mechanism which is used for TORS internal communication. Objects can subscribe to topics and publish messages on the bus.
tors.nodecom tors.nodecom.mail taors.nodecom.network tors.nodecom.file	An abstraction of various types of communication protocols between network nodes. These protocols are used to send and receive TravelingObjects. In the examined version, network and file based protocols have been defined. e-mail based node communication is planned, so the tors.nodecom.mail package is empty.
tors.nameserver	A nameserver to provide a mapping between TO names and their location in the network.
tors.dataaccess	A mechanism to read and write data.
tors.core	The core of the TORS. Contains an Objectmanager, a modified ClassLoader, a TO-Generator, a modified SecurityManager and other utilities used by the system.
ToDoExample TestExample	Containing examples, tests and a simple help desk application which allows to send and receive various types of problem tickets.

12.3.1.2 Estimated properties of the original project

Table 18: TOS: original estimations

Package	FragmentSize	PublicFactor	Coupling	InvCoupling	Coupling hardness	Coupling Tension	Cohesion
TorsServices	1900	0,50	350	0	-200	0	36,36
tors.view	2500	0,40	450	0	-400	0	111,82
tors.to	700	0,80	300	50	-300	250	0,91
tors.resources	20	0,40	0	20	-50	0	0,00
tors.objectbus	150	0,80	30	450	200	50	4,55
tors.nodecom.network	1200	0,50	980	1	-900	0	3,00
tors.nodecom.mail	0	0,00	0	0	0	0	0,00
tors.nodecom.file	1100	0,55	950	1	-500	0	2,09
tors.nodecom	1300	0,60	600	1200	700	900	9,09
tors.nameserver	200	0,60	100	100	50	200	0,00
tors.dataaccess	500	0,65	50	600	550	120	0,45
tors.core	1700	0,60	350	300	300	600	10,91
tors	0	0,00	0	0	0	0	0,00
ToDoExample	50	0,40	100	0	-30	0	0,00
TestExample	95	0,80	50	0	-30	0	4,09
AVERAGE	761,00	0,51	287,33	181,47	-40,67	141,33	12,22

12.3.1.3 Measurement results of the original project

Table 19: TOS: original measurements

Package	FragmentSize	PublicFactor	Coupling	InvCoupling	Coupling hardness	Coupling Tension	Cohesion
TorsServices	1767	0,58	301	0	-301	0	37,45
tors.view	2495	0,41	499	0	-499	0	94,31
tors.to	975	0,72	280	93	-187	186	2,75
tors.resources	30	0,36	20	0	-20	0	0,00
tors.objectbus	170	0,93	86	394	308	172	1,29
tors.nodecom.network	1230	0,58	912	0	-912	0	0,00
tors.nodecom.mail	0	0,00	0	0	0	0	0,00
tors.nodecom.file	1180	0,49	980	0	-980	0	3,00
tors.nodecom	1394	0,60	450	1225	775	900	9,43
tors.nameserver	365	0,56	143	176	33	286	0,75
tors.dataaccess	574	0,71	82	898	816	164	27,00
tors.core	1580	0,63	436	275	-161	550	8,67
tors	0	0,00	0	0	0	0	0,00
ToDoExample	88	0,22	14	0	-14	0	0,00
TestExample	122	0,75	31	0	-31	0	1,50
AVERAGE	798,00	0,50	282,27	204,07	-78,20	150,53	12,41

12.3.1.4 Heuristic violations in original project

Table 20: TOS heuristic violations

MH06	Subclass: [tors.core.RuntimeVersion] of [tors.to.Version] adds less than 5 perCent to its superclass.	*Remove Class*
MH02	Association: [TorsServices.ToNameServer]..mFrameModeNext Is PublicAccess Association: [TorsServices.ToNameServer]..toName Is PublicAccess	*Stronger Protection*
	Attribute: [tors.nodecom.network.NetworkPortServer].currentPort is ProtectedAccess Attribute: [tors.nodecom.network.NetworkPortServer].basePort is ProtectedAccess	*Stronger Protection*
	Attribute: [tors.to.TravellingObject].fixed is ProtectedAccess Attribute: [tors.to.TravellingObject].master is ProtectedAccess Attribute: [tors.to.TravellingObject] returnToSender is ProtectedAccess Attribute: [tors.to.TravellingObject].isRunning is ProtectedAccess Attribute: [tors.to.TravellingObject].activeThreads is ProtectedAccess	*Stronger Protection*
	Association: [TorsServices.Problem]..Solutions Is PublicAccess Association: [TorsServices.Problem]..Solvers Is PublicAccess	*Stronger Protection*
	Association: [tors.core.TOClassLoader]..mClassLoaderTable Is PublicAccess Association: [tors.core.TOClassLoader]..mLoader1 Is PublicAccess Association: [tors.core.TOClassLoader]..mLoader2 Is PublicAccess Association: [tors.core.TOClassLoader]..beanLoader Is ImplementationAccess	*Stronger Protection*
	Attribute: [tors.nodecom.network.NetworkPortServer].currentPort is ProtectedAccess Attribute: [tors.nodecom.network.NetworkPortServer].basePort is ProtectedAccess Association: [tors.nodecom.network.NetworkPortServer]..objectBus Is ProtectedAccess	*Stronger Protection*
	Association: [TorsServices.Ticket]..objectBus Is ImplementationAccess	*Stronger Protection*
	Association: [tors.objectbus.ObjectBusException]..exceptionMessage Is ProtectedAccess Association: [tors.objectbus.ObjectBusEvent]..eventDescription Is ProtectedAccess	*Stronger Protection*
MH14	Redundant property: [MainObjectBus objectBus] occurs in: [ToNameServer] as [Role]: [NodeManager] as [Role]: [LogicalNode] as [Role]: [NameServer] as [Role]: [TORSSecurityManager] as [Role]:	*Singleton redesign*
	Redundant property: [DataAccessor dataAccessor] occurs in: [NodeManager] as [Role]: [NameServer] as [Role]: [PackageGenerator] as [Role]: [TOClassLoader] as [Role]: [NetworkPortServer] as [Role]: [ObjectExitManager] as [Role]: [FileObjectEntry] as [Role]: [NetworkObjectEntry] as [Role]: [TravellingObject] as [Role]: [ObjectManager] as [Role]: [FileListenServer] as [Role]: [ObjectExit] as [Role]:	*Singleton redesign*
	Redundant property: [String owner] occurs in: [FileObjectEntry] as [Role]: [NetworkObjectEntry] as [Role]: [NameServerEntry] as [Role]:	*Missing base class*
	Redundant property: [NodeManager nodeManager] occurs in: [AddNodeDialog] as [Role]: [MoveObjectDialog] as [Role]: [MainDialog] as [Role]: [NameServer] as [Role]: [LogicalNodeDialog] as [Role]: [NetworkPortServer] as [Role]: [ObjectExitManager] as [Role]: [FileObjectEntry] as [Role]: [NetworkObjectEntry] as [Role]: [ObjectManager] as [Role]: [FileListenServer] as [Role]:	*Singleton redesign*
	Redundant property: [Properties systemProperties] occurs in: [NodeManager] as [Role]: [MainDialog] as [Role]: [FileAccessor] as [Role]: [NetworkPortServer] as [Role]: [ObjectExitManager] as [Role]: [DataAccessor] as [Role]: [FileObjectEntry] as [Role]: [NetworkObjectEntry] as [Role]: [ObjectManager] as [Role]: [FileListenServer] as [Role]: [ObjectExit] as [Role]:	*Singleton redesign*
	Redundant property: [ObjectManager objectManager] occurs in: [StartObjectDialog] as [Role]: [ActiveObjectsDialog] as [Role]: [MoveObjectDialog] as [Role]: [MainDialog] as [Role]: [FileObjectEntry] as [Role]: [TravellingObject] as [Role]: [FileListenServer] as [Role]:	*Singleton redesign*
	Redundant property: [boolean isSended] occurs in: [FileObjectExit] as [Attribute]: [ObjectExitManager] as [Attribute]: [NetworkObjectExit] as [Attribute]: Redundant property: [boolean isStillSending] occurs in: [FileObjectExit] as [Attribute]: [NetworkObjectExit] as [Attribute]:	*Base class move*
	Redundant property: [TOClassLoader classLoader] occurs in: [PackageGenerator] as Role: [FileAccessor] as [Role]: [TOClassLoader] as [Role]: [NetworkPortServer] as [Role]: [ObjectExitManager] as [Role]: [Ticket] as [Role]: [DataAccessor] as [Role]: [FileObjectEntry] as [Role]: [NetworkObjectEntry] as [Role]: [TORSDialog] as	*Singleton redesign*

	[Role]: [TravellingObject] as [Role]: [TestEditor] as [Role]: [ObjectManager] as [Role]: [FileListenServer] as [Role]: [ToCommand] as [Role]: [ObjectExit] as [Role]:	
MH17	Operation: [tors.nodecom.ObjectExit].run occurs in all of the subclasses of [tors.nodecom.ObjectExit]. It should be moved To [tors.nodecom.ObjectExit]	*Base class move*
	Operation: [tors.nodecom.ObjectExit].stop occurs in all of the subclasses of [tors.nodecom.ObjectExit]. It should be moved To [tors.nodecom.ObjectExit]	*Base class move*
	Operation: [tors.nodecom.ObjectExit].getIsSended occurs in all of the subclasses of [tors.nodecom.ObjectExit]. It should be moved To [tors.nodecom.ObjectExit] Operation: [tors.nodecom.ObjectExit].getIsStillSending occurs in all of the subclasses of [tors.nodecom.ObjectExit]. It should be moved To [tors.nodecom.ObjectExit]	*Base class move*
MH25	[tors.core.TOClassLoader] has a direct recursive Association [] [tors.core.TOClassLoader] has a direct recursive Association []	*Not used*
MH24	[tors.nameserver.NameServer] depends on a softer class [tors.nameserver.NameServerEntry] [tors.nameserver.NameServer] depends on a softer class [tors.nameserver.Nameable] tors.core.ObjectManager] depends on a softer class [tors.core.TOClassLoader] [tors.core.ObjectManager] depends on a softer class [tors.nameserver.NameServer] [tors.core.ObjectManager] depends on a softer class [tors.to.TravellingObject] [(tors.nodecom.ObjectExitManager] depends on a softer class [tors.to.TOInterface] [tors.objectbus.MainObjectBus] depends on a softer class [tors.objectbus.Subject] [tors.objectbus.MainObjectBus] depends on a softer class [tors.objectbus.Subscriber] [tors.to.TravellingObject] depends on a softer class [tors.core.TOClassLoader] [tors.nodecom.ObjectExit] depends on a softer class [tors.nameserver.NameServer]	*Singleton redesign (situation 1)*
	[TorsServices.Ticket] depends on a softer class [tors.to.TOInterface] [TorsServices.ToCommand] depends on a softer class [tors.to.TOInterface] [tors.nodecom.ObjectExit] depends on a softer class [tors.to.TravellingObject]	*Singleton redesign (situation 2)*
	[TorsServices.ToCommand] depends on a softer class [TorsServices.ToCommandUI]	*UI dependency*

12.3.1.5 Measurement results of transformed project

Table 21: TOS: transformed project measures

Package	FragmentSize	PublicFactor	Coupling	InvCoupling	Coupling hardness	Coupling Tension	Cohesion
TorsServices	2581	0,41	359	0	-359	0	39,55
tors.view	2736	0,37	508	0	-508	0	119,67
tors.to	1152	0,65	282	83	-199	166	6,50
tors.resources	30	0,50	48	0	-48	0	0,00
tors.objectbus	402	0,72	86	393	307	172	1,57
tors.nodecom.network	1137	0,64	574	0	-574	0	0,00
tors.nodecom.mail	0	0,00	0	0	0	0	0,00
tors.nodecom.file	827	0,45	381	0	-381	0	4,50
tors.nodecom	1458	0,59	455	683	228	910	9,29
tors.nameserver	365	0,56	143	176	33	286	0,00
tors.dataaccess	580	0,70	88	893	805	176	27,00
tors.core	1637	0,61	424	271	-153	542	7,22
tors	0	0,00	0	0	0	0	0,00
ToDoExample	122	0,16	14	0	-14	0	0,00
TestExample	140	0,65	30	0	-30	0	1,50
AVERAGE	877,80	0,47	226,13	166,60	-59,53	150,13	14,45

12.3.2 Empiric Evaluation

12.3.2.1 Evaluation of Measurements

To examine the correlation between the estimated and the measured values, the correlation coefficient (Triola 1999) has been used. The values of the coefficient range from -1 to +1. The value of +1 denotes a perfect positive correlation and -1 a prefect negative correlation. If there is no correlation, the value is 0. The correlation efficient for the MeTHOOD measures and evaluation of TOS are shown in Table 22,

Table 22: TOS: correlation efficients

FragmentSize	PublicFactor	Coupling	InvCoupling	Coupling hardness	Coupling Tension	Cohesion
0,993	0,958	0,982	0,978	0,926	0,98527	0,959

All estimated values are highly correlated to the measured values. Table 23 shows the absolute differences between the estimated and measured values.

Table 23: TOS: difference of estimated and measured

Package	FragmentSize	PublicFactor	Coupling	InvCoupling	Coupling hardness	Coupling Tension	Cohesion
TorsServices	133	-0,08	49	0	101	0	-1,09
tors.view	5	-0,01	-49	0	99	0	17,51
tors.to	-275	0,08	20	-43	-113	64	-1,84
tors.resources	-10	0,04	-20	20	-30	0	0,00
tors.objectbus	-20	-0,13	-56	56	-108	-122	3,26
tors.nodecom.network	-30	-0,08	68	1	12	0	3,00
tors.nodecom.mail	0	0,00	0	0	0	0	0,00
tors.nodecom.file	-80	0,06	-30	1	480	0	-0,91
tors.nodecom	-94	0,00	150	-25	-75	0	-0,34
tors.nameserver	-165	0,04	-43	-76	17	-86	-0,75
tors.dataaccess	-74	-0,06	-32	-298	-266	-44	-26,55
tors.core	120	-0,03	-86	25	461	50	2,24
tors	0	0,00	0	0	0	0	0,00
ToDoExample	-38	0,18	86	0	-16	0	0,00
TestExample	-27	0,05	19	0	1	0	2,59
AVERAGE	-37,00	0,0049	5,07	-22,60	37,53	-9,20	-0,19

The variations between these values are very low. In most cases, the measured values define the same ordering of the components as the estimated values.

12.3.2.2 Evaluation of heuristics: Specific comments to list of heuristic violations

MH02:

Stronger Protection

The project contains many attributes and associations which have public, protected or implementation access. Nearly all of them can be transformed to private.

MH06:

Remove Class

Some subclasses add very little functionality to their base classes. An example is *RuntimeVersion* which only refines the constructor with an hard coded version number. These classes can be removed.

MH14:

Singleton redesign

The project contains a set of classes which have only one instance. These classes are *MainObjectBus, TOClassLoader, ObjectManager, DataAccessor, Properties, NodeManager* and *NameServer*. Classes that use them carry instance variables for instances which are initialized in the constructor. This leads to a situation in which many methods need to specify parameters with these classes to pass their instances through the system. The member variables, the initialization in the constructor and the method parameters with these classes are unnecessary and can be removed if the classes are transformed into Singletons.

Base class move

Many member variables, e.g. *isSended* and *isStillSending* in subclasses of *ObjectExit*, occur in all subclasses of their base class. These variables (and corresponding methods using it) can be moved to the base class, which simplifies all of the subclasses.

Missing base class

Some member variables, e.g. *owner* in *FileObjectEntry* and *NetworkObjectEntry*, occur in different classes. A simple analysis of this situation shows that in some cases, base classes are missing. An example for such a missing base class is *ObjectEntry* with the subclasses *FileObjectEntry* and *NetworkObjectEntry*, which would correspond to *ObjectExit* with the subclasses *FileObjectExit* and *NetworkObjectExit*.

MH17:

Base class move

Some implementations of methods in subclasses of a base class are identical, e.g. *getIsStillSending, stop, run* in *ObjectExit*. These methods can be moved to the superclass.

MH24:

Singleton redesign

Many potential singleton classes are recognized (e.g, *NameServer, ObjectManager MainObjectBus*) as coupling-hard because they are referenced by many other classes as member and parameter types (situation 1). Other classes, e.g. TOInterface, are recognized as soft because they reference potential singleton classes, e.g. MainObjectBus and ObjectManager (situation 2). Both types of situations would be removed using the singleton redesign proposed above. The conflict in this heuristic is another (interesting) consequence of the singleton redesign problem.

UI dependency

Some business classes (e.g. TOCommand, an editable command class for other TOs) depend directly on their user interface. This is because the TravelingObjects paradigm allows to bundle the user interface and the business classes. Thus, this heuristic violation is no design flaw – it only expresses the differences in the paradigm.

MH25:

Not used

Some elements in the schema are not used (properties classes or associations) e.g. *mLoader1* and *mLoader2* in *TOClassLoader*, and thus can be removed.

12.3.2.3 Evaluation of heuristic based transformation

Table 24: TOS: difference between original and transformed

Package	FragmentSize	PublicFactor	Coupling	InvCoupling	Coupling hardness	Coupling Tension	Cohesion
TorsServices	814	-0,17	58	0	-58	0	2,09
tors.view	241	-0,04	9	0	-9	0	25,36
tors.to	177	-0,07	2	-10	-12	-20	3,75
tors.resources	0	0,14	28	0	-28	0	0,00
tors.objectbus	232	-0,21	0	-1	-1	0	0,29
tors.nodecom.network	-93	0,06	-338	0	338	0	0,00
tors.nodecom.mail	0	0,00	0	0	0	0	0,00
tors.nodecom.file	-353	-0,03	-599	0	599	0	1,50
tors.nodecom	64	-0,02	5	-542	-547	10	-0,14
tors.nameserver	0	0,00	0	0	0	0	-0,75
tors.dataaccess	6	-0,01	6	-5	-11	12	0,00
tors.core	57	-0,02	-12	-4	8	-8	-1,44
tors	0	0,00	0	0	0	0	0,00
ToDoExample	34	-0,06	0	0	0	0	0,00
TestExample	18	-0,10	-1	0	1	0	0,00
AVERAGE	79,80	-0,03	-56,13	-37,47	18,67	-0,40	2,04

For the situations described above, the default transformations have executed the proposed changes: *Base class move*. For the other proposed transformations, especially *Singleton redesign*, manual work is required.

12.4 iBus

12.4.1 Project Assessment

iBus is a message based middleware for Java-applications. It provides a publish/subscribe messaging mechanism for Java objects. iBus allows to develop message producers which are independent from the consumers of the messages. It includes a so-called „quality of service"-framework, allowing to „attach" qualities of service, such as failure detection and reliable multicast, to communication channels. iBus has a flexible architecture which can easily be extended. Furthermore, it allows to communicate using Java objects. We have examined version 0.4 beta of iBus.

12.4.1.1 Description of the project architecture

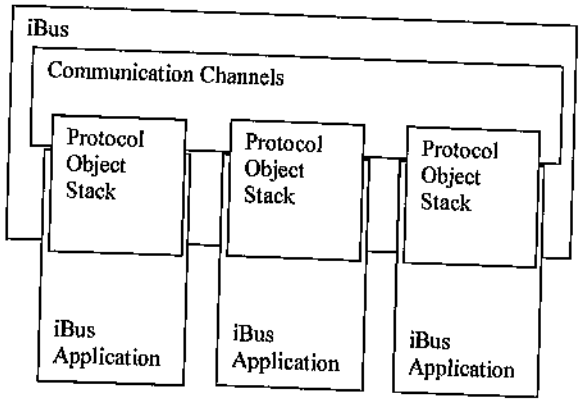


Figure 56: High-Level Architecture of iBus

At the core of iBus architecture are *communication channels* which encapsulates communication details from communicating channel members.

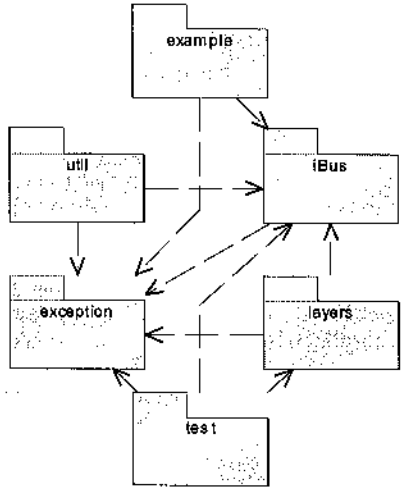


Figure 57: iBus components

An important part of the channel description is the *quality of service (QOS)*, which is realized by a stack of so-called *protocol objects*. A protocol object is an object used to handle different kinds of communication tasks, e.g., sending and receiving UDP datagrams, failure detection and membership etc. Users of iBus can configure different kinds of protocol object *stacks* and even plug their own protocol objects into the framework. iBus consist of 78 classes and interfaces.
Figure 57 illustrates the iBus packages and their dependencies. The packages have the following functions:

iBus.util	Simple talkers and listeners which post simple messages on the bus.
iBus.test	A set of test classes which are used to test different aspects of the system.
iBus.layers	Protocol objects which can be used to configure communication links.
iBus.exception	A set of exception classes.
iBus.example	A simple „hello-world"-like example that uses the bus.
iBus	Main iBus package includes the implementation of a stack of protocol objects, the iBus-URL, different kinds of events and the protocol for messages and postings.

12.4.1.2 Estimated properties of the original project

Table 25: iBus: original estimations

	FragmentSize	PublicFactor	Coupling	InvCoupling	Coupling hardness	Coupling Tension	Cohesion
iBus.util	775	0,60	3500	100	-800	0	0,00
iBus.test	6425	0,70	5200	0	-1500	0	0,00
iBus.layers	8738	0,40	4000	500	-3000	20	8,70
iBus.exception	900	1,00	0	6300	-200	0	0,00
iBus.example	200	0,90	5200	0	-60	0	0,00
iBus	7710	0,70	200	6000	4000	80	166,67
AVERAGE	4124,67	0,72	3016,67	2150,00	-260,00	16,67	29,23

12.4.1.3 Measurement results of the original project

Table 26: iBus: original measurements

	FragmentSize	PublicFactor	Coupling	InvCoupling	Coupling hardness	Coupling Tension	Cohesion
iBus.util	602	0,67	208	0	-208	0	0,00
iBus.test	2276	0,67	1054	0	-1054	0	0,22
iBus.layers	8742	0,63	5160	0	-5160	0	16,04
iBus.exception	30	1,00	10	0	-10	0	0,00
iBus.example	100	1,00	8	0	-8	0	0,00
iBus	6726	0,66	67	6298	6231	134	165,38
AVERAGE	3079,33	0,77	1084,50	1049,67	-34,83	22,33	30,27

12.4.1.4 Heuristic violations in original project

Table 27: iBus heuristic violatons

MH06	Subclass: [iBus.ProblemEvent] of [iBus.Event] adds less than 5 perCent to its superclass.	*Remove Class*
	Subclass: [iBus.test.FIFO_test_checker] of [iBus.ProtocolObject] adds less than 5 perCent to its superclass. Subclass: [iBus.layers.BADNET] of [iBus.ProtocolObject] adds less than 5 perCent to its superclass. Subclass: [iBus.layers.SEQCHK] of [iBus.ProtocolObject] adds less than 5 perCent to its superclass. Subclass: [iBus.layers.LOCALBUS] of [iBus.ProtocolObject] adds less than 5 perCent to its superclass. Subclass: [iBus.layers.FIFO] of [iBus.ProtocolObject] adds less than 5 perCent to its superclass. :Subclass: [iBus.layers.IPMCAST] of [iBus.ProtocolObject] adds loss than 5 perCent to its superclass.	*Abstract method call*
MH02	Association: [iBus.util.TalkerMembership]..log_ Is ImplementationAccess Association: [iBus.test.IPMCAST_test]..log_ Is ImplementationAccess Attribute: [iBus.layers.FRAG].fragSize_ is ProtectedAccess Association: [iBus.layers.FRAG]..log_ Is ImplementationAccess Association: [iBus.layers.NAK]..log_ Is ImplementationAccess Attribute: [iBus.iBusURL].ipOctets is PublicAccess Association: [iBus.iBusURL]..log_ Is ImplementationAccess Association: [iBus.util.listener]..log_ Is ImplementationAccess Association: [iBus.Stack]..membershipItf Is PublicAccess	*Configurable Logging*

	Association: [iBus.Stack]..log_ Is ImplementationAccess Association: [iBus.test.Pusher]..log_ Is ImplementationAccess Association: [iBus.StackID]..log_ Is ImplementationAccess Association: [iBus.Event]..log_ Is ImplementationAccess Association: [iBus.Event]..log_ Is ImplementationAccess Association: [iBus.util.PerfListenerReceiver]..log_ Is ImplementationAccess Association: [iBus.test.Posting_test]..log_ Is ImplementationAccess Association: [iBus.util.talker]..log_ Is ImplementationAccess Association: [iBus.test.REACH_test]..log_ Is ImplementationAccess Association: [iBus.test.NAKReceiver]..log_ Is ImplementationAccess Association: [iBus.test.MessageEvent_test]..log_ Is ImplementationAccess Association: [iBus.test.LOCALBUS_test]..log_ Is ImplementationAccess Association: [iBus.util.ListenerReceiver]..log_ Is ImplementationAccess Association: [iBus.MessageEvent]..log_ Is ImplementationAccess Association: [iBus.test.LOCALBUS_test_rcv]..log_ Is ImplementationAccess Association: [iBus.test.FIFO_test]..log_ Is ImplementationAccess Association: [iBus.util.perf_listener]..log_ Is ImplementationAccess Attribute: [iBus.test.NAK_test].resubscribe_ is ProtectedAccess Association: [iBus.test.NAK_test]..log_ Is ImplementationAccess Association: [iBus.layers.REACH]..log_ Is ImplementationAccess Attribute: [iBus.ProtocolObject].defaultPort_ is ProtectedAccess Association: [iBus.ProtocolObject]..log_ Is ImplementationAccess Association: [iBus.ProtocolObject]..log_ Is ImplementationAccess Association: [iBus.ProtocolObject]..log_ Is ImplementationAccess Association: [iBus.ProtocolObject]..log_ Is ImplementationAccess Association: [iBus.ProtocolObject]..log_ Is ImplementationAccess Association: [iBus.test.FRAG_test]..log_ Is ImplementationAccess Association: [iBus.ProtocolObject]..log_ Is ImplementationAccess	
	Association: [iBus.layers.Source]..socket_ Is ProtectedAccess Association: [iBus.layers.Source]..receiver_ Is ProtectedAccess Association: [iBus.layers.Source]..pt2ptAddress_ Is ProtectedAccess Association: [iBus.layers.Source]..master_ Is ProtectedAccess Association: [iBus.test.REACH_test_mon]..views_ Is ProtectedAccess Attribute: [iBus.test.SerializableTest].i_ is PublicAccess Association: [iBus.test.SerializableTest]..str_ Is PublicAccess	*Stronger Protection*
	Association: [tors.core.TOClassLoader]..mClassLoaderTable Is PublicAccess Association: [tors.core.TOClassLoader]..mLoader1 Is PublicAccess Association: [tors.core.TOClassLoader]..mLoader2 Is PublicAccess Association: [tors.core.TOClassLoader]..beanLoader Is ImplementationAccess	*Stronger Protection*
MH08	Base Class [iBus.ProtocolObject] has dependency from a derived class [iBus.Stack] MH08:Base Class [iBus.ProtocolObject] has dependency from a derived class [iBus.Stack]	*Composite*
MH14	Redundant property: [Log log_] occurs in: [perf_talker] as [Role]: [PostingReceiver] as [Role]: [TalkerMembership] as [Role]: [IPMCAST_test] as [Role]: [FRAG] as [Role]: [NAK] as [Role]: [iBusURL] as [Role]: [listener] as [Role]: [Stack] as [Role]: [Pusher] as [Role]: [FRAG_test] as [Role]: [StackID] as [Role]: [Event] as [Role]: [PerfListenerReceiver] as [Role]: [Posting_test] as [Role]: [talker] as [Role]: [REACH_test] as [Role]: [NAKReceiver] as [Role]: [MessageEvent_test] as [Role]: [LOCALBUS_test] as [Role]: [ListenerReceiver] as [Role]: [MessageEvent] as [Role]: [LOCALBUS_test_rcv] as [Role]: [FIFO_test] as [Role]: [perf_listener] as [Role]: [NAK_test] as [Role]: [REACH] as [Role]: [ProtocolObject] as [Role]:	*Missing Base Class Method*
	Redundant property: [iBusURL channel_] occurs in: [IncomingChecker] as [Role]: [View] as [Role]: [FIFO_SenderInfo] as [Role]: [Event] as [Role]: [SenderInfo] as [Role]: [SEQCHK_SenderInfo] as [Role]: [REACH_GroupInfo] as [Role]: [ChannelInfo] as [Role]:	*Missing Abstraction*
	Redundant property: [int hbInterval_] occurs in: [NAK] as [Attribute]: [REACH] as [Attribute]: MH14:Redundant property: [Thread hbThread_] occurs in: [NAK] as [Role]: [REACH] as [Role]:	*Missing base class*
	Redundant property: [boolean isListener_] occurs in: [ChannelMember] as [Attribute]:	*Duplicate property*

	[REACH_HbeatParam] as [Attribute]: [REACH_GroupMember] as [Attribute]: Redundant property: [boolean isTalker_] occurs in: [ChannelMember] as [Attribute]: [REACH_HbeatParam] as [Attribute]: [REACH_GroupMember] as [Attribute]: MH14:Redundant property: [long lastInOrderSeq_] occurs in: [FIFO_SenderInfo] as [Attribute]: [SEQCHK_SenderInfo] as [Attribute]: MH14:Redundant property: [iBusURL sender_] occurs in: [FIFO_SenderInfo] as [Role]: [Event] as [Role]: [SenderInfo] as [Role]: [SEQCHK_SenderInfo] as [Role]:	
MH25	[iBus.ProtocolObject] has a direct recursive Association [] [iBus.ProtocolObject] has a direct recursive Association []	*Composite*

12.4.1.5 Measurement results of transformed project

Table 28: iBus: transformed project measurements

	FragmentSize	PublicFactor	Coupling	InvCoupling	Coupling hardness	Coupling Tension	Cohesion
iBus.util	702	0,59	248	0	-248	0	0,00
iBus.test	1908	0,52	578	0	-578	0	0,18
iBus.layers	6973	0,53	3864	3	-3861	6	15,67
iBus.exception	30	1,00	10	0	-10	0	0,00
iBus.example	104	1,00	8	0	-8	0	0,00
iBus	8508	0,78	81	4587	4506	162	285,13
AVERAGE	3037,50	0,74	798,17	765,00	-33,17	28,00	50,16

12.4.2 Empiric Evaluation

12.4.2.1 Evaluation of Measures

The correlation efficients of the Ibus measures and estimations are shown in Table 29.

Table 29: IBus: correlation efficients

FragmentSize	PublicFactor	Coupling	InvCoupling	Coupling hardness	Tension	Cohesion
0,916	0,880	0,303	0,607	0,983	0,968	0,999

Most of the figures indicate high correlation. The correlation of the coupling and inverse coupling has to be examined closer.

Table 30: iBus: difference between estimated and measured

	FragmentSize	PublicFactor	Coupling	InvCoupling	Coupling hardness	Coupling Tension	Cohesion
iBus.util	-173	0,07	-3292	-100	592	0	0,00
iBus.test	-4149	-0,03	-4146	0	446	0	0,22
iBus.layers	4	0,23	1160	-500	-2160	-20	7,35
iBus.exception	-870	0,00	10	-6300	190	0	0,00
iBus.example	-100	0,10	-5192	0	52	0	0,00
iBus	-984	-0,04	-133	298	2231	54	-1,29

	FragmentSize	PublicFactor	Coupling	InvCoupling	Coupling hardness	Coupling Tension	Cohesion
AVERAGE	-1045,33	0,05	-1932,17	-1100,33	225,17	5,67	1,05

The absolute differences (shown in Table 30) show that the measured values are lower almost everywhere. The reason for this is that the source code contains objects and relationships that are not visible in the schema. In most cases, the measured values define the same ordering of the components as the estimated values. Some rank movements occur for fields with low differences between measured and estimated values.

Using the absolute and the rank differences of the measured and estimated values we are able to identify errors based on the selected evaluation process (measuring the schema and estimating the reengineered source code). The following differences can be explained and do not represent "mistakes" of the measures:

1. *FragmentSize of iBus test and iBus exception*: The test classes contain few test methods that test many aspects of the system. In each case, method local variables are used which are not visible inside the model. Therefore, the author must make an estimation that is higher than the measured values. The same is true for the exception classes.
2. *Coupling of iBus.test and iBus.example*: Both packages contain classes which use few test and example methods which use local variables that are visible only inside the methods. This dependency is not visible in the schema because the reverse engineered schema does not contain this type of method dependencies. Therefore, the author must make a higher estimation of coupling for both packages.
3. *Inverse Coupling of iBus.exception*. Dependencies on exceptions are not correctly recognized by this evaluation (see 12.1.2). Therefore, the estimation of inverse coupling and coupling tension must be higher than the measured value.

Table 31 shows the correlation efficient for the measured and estimated values where the five cases above have been removed.

Table 31: IBus: corrected correlation efficients

Package	FragmentSize	PublicFactor	Coupling	InvCoupling	Coupling hardness	Coupling Tension	Cohesion
TorsServices	0,995	0,880	0,677	0,997	0,983	0,968	0,999

12.4.2.2 Evaluation of heuristics: Specific comments to the list of heuristic violations

MH06:

Remove Class

The ID and the class type are both used to differentiate between different types of events. This constellation can be reduced to classification by ID and the class *ProblemEvent* could be removed.

Abstract method call

The design of these classes has the following problem:
overwriten abstract methods are directlycalled by client objects. This leads to code duplication in the subclass methods. Providing a concrete implementation in the superclass would make it possible to remove this duplicated code.

Configurable Logging

Many classes have *log_* Objects. All of them are static. The idea is to implement a per-component log filtering. It would be very hard to change interface or implementation of the Log class. A configurable base class logging could improve this problem.

MH02:

Stronger Protection

This classes should use get- and setters. The weak encapsulation is not nessesary.

MH08:

Composite

This constellation could be transformed to a composite. This would remove the dependency.

MH14:

Missing Base Class Method

This constellation can be simplified, e.g. using log method in ProtocolObject with name of protocol object.

Missing base class

A baseclass *SenderInfo* with *channel* and *sender* is missing. At least for *SEQCHK_SenderInfo* and *FIFO_SenderInfo*. This flaw leads to copied code.

Missing Abstraction

Both *NAK* and *REACH* maintain a heartbeat Thread with a heartbeat interval. Introduction of a *HeardBeat* abstraction would resolve this problem.

Duplicate property

REACH_GroupMember should use the properties of *ChannelMember*, but does not use it. In the other cases, similar problems occur.

MH25:

Composite

The solution of the composite problem in MH08 would resolve this problem.

12.4.2.3 Evaluation of heuristic based transformation

Table 32: iBus: difference between original and transformed

	FragmentSize	PublicFactor	Coupling	InvCoupling	Coupling hardness	Coupling Tension	Cohesion
iBus.util	100	-0,07	40	0	-40	0	0,00
iBus.test	-368	-0,15	-476	0	476	0	-0,05
iBus.layers	-1769	-0,10	-1296	3	1299	6	-0,38
iBus.exception	0	0,00	0	0	0	0	0,00
iBus.example	4	0,00	0	0	0	0	0,00
iBus	1782	0,12	14	-1711	-1725	28	119,75
AVERAGE	-41,83	-0,03	-286,33	-284,67	1,67	5,67	19,89

12.5 JDK 1.1.5 class library

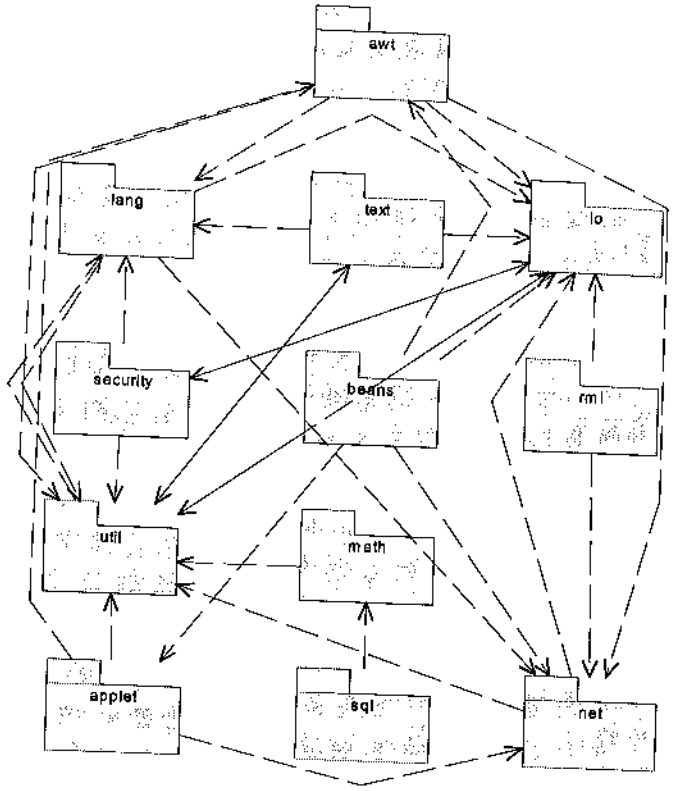


Figure 58: JDK 1.1.5 components

The Java Development Kit (JDK) is a development environment for writing Java applets and applications. The JDK class library is a central element of the JDK. It is a well-known class library containing the core classes of the Java platform. The examination of the JDK class library has been included in this examination because most readers of this thesis are familiar with the library and can use it as a reference and a source for comparison.

12.5.1 Project Assessment

12.5.1.1 Description of the project architecture

The JDK class library is no application. Therefore, it has no project architecture of its own. It is subdivided into twelve packages which are the base library for all Java applications.

Figure 54 illustrates the packages contained in the JDK, and their dependencies. The packages have the following functions:

java.awt.datatransfer	Includes mechanism to transfer data on the user interface, e.g., a clipboard.
java.security	Includes security related classes, e.g., a definition of a group and a security manager.
java.rmi.server	Includes base classes needed to build an RMI (Remote Method Invocation) server.
java.util	Includes utilities used everywhere, e.g., Vector, Date, Hashtable.
java.awt.peer	Includes a large set of interfaces of Peers. These interfaces are used to encapsulate user interface elements.
java.security.interfaces	Includes interfaces for specific security algorithms.
java.security.acl	Includes the definition of an access control list which allows to specify permissions
java	Empty base package.
java.lang.reflect	Includes mechanisms needed to dynamically invoke methods, using their names.
java.awt.event	Includes a large set of classes which represent different kinds of user interface events, e.g., MouseEvent.
sun	Empty base package for Sun classes.
java.io	A large set of classes which handle all forms of stream based data in and output., e.g., FileInputStream and FileReader.
java.net	Includes mechanisms to use network connections, e.g., Socket and HttpURLConnection.
java.awt.image	Includes different classes to handle images, e.g. a ColorModel and a PixcelGrabber.
java.rmi.registry	Includes a registry for RMI servers.
java.rmi	Includes mechanisms needed to develop RMI applications.
java.sql	Includes the JDBC (Java Database Connectivity) interface to access relational databases.
java.awt	Includes a framework (e.g. the classes Frame, Window and Menue) for building Java based user interfaces.

java.applet	Includes the applet base class. An applet is a program that is intended to be embedded inside another application.
java.math	Includes different types used from calculations, e.g. BigDecimal.
java.util.zip	Includes mechanisms to handle compresses (zip) Files.
java.rmi.dgc	Includes a base mechanism for a distributed garbage collector (DGC), e.g., a unique identification across VMs (Virtual Machines).
sun.misc	Includes a few classes provided by Sun.
java.beans	Includes different classes to handle Java Beans, e.g., Property and Method descriptors.
java.text	Includes a large set of classes to handle parse and format text data.
java.text.resources	Includes a large set of classes needed to implement locale dependent formatting for a standard set of locales.
java.lang	Includes the fundamental Java language elements, e.g. the classes Object and Sting.

12.5.1.2 Measurement results of the original project

Table 33: JDK: original measurements

	FragmentSize	PublicFactor	Coupling	invCoupling	Coupling hardness	Coupling Tension	Cohesion
java.awt.datatransfer	838	0,78	309	1	-308	2	13,00
java.security	5149	0,80	2595	52	-2543	104	31,50
java.rmi.server	3167	0,73	1538	15	-1523	30	21,04
java.util	10091	0,76	1579	12953	11374	3158	127,50
java.awt.peer	1912	0,00	35	27	-8	54	1,44
java.security.interfaces	566	1,00	27	0	-27	0	4,00
java.security.acl	837	1,00	529	0	-529	0	7,25
java	0	0,00	0	0	0	0	0,00
java.lang.reflect	3220	0,69	788	176	-612	352	0,43
java.awt.event	14404	0,93	1941	149	-1792	298	189,57
sun	0	0,00	0	0	0	0	0,00
java.io	12479	0,57	973	3108	2135	1946	74,78
java.net	6074	0,61	1298	88	-1210	176	50,52
java.awt.image	2948	0,78	59	16	-43	32	76,93
java.rmi.registry	136	0,73	9	1	-8	2	1,00
java.rmi	4524	0,93	1403	635	-768	1270	130,79
java.sql	3448	0,46	716	0	-716	0	29,33
java.awt	197437	0,85	1692	13284	11592	3384	2126,80
java.applet	12129	0,91	11410	99	-11311	198	1,75
java.math	8210	0,80	39	6	-33	12	341,00
java.util.zip	3987	0,64	893	0	-893	0	86,16
java.rmi.dgc	167	0,34	21	0	-21	0	4,67

	FragmentSize	PublicFactor	Coupling	InvCoupling	Coupling hardness	Coupling Tension	Cohesion
sun.misc	1	0,00	0	1	1	0	0,00
java.beans	8754	0,58	1735	0	-1735	0	45,04
java.text	20602	0,76	972	0	-972	0	213,78
java.text.resources	24331	0,68	12893	0	-12893	0	112,40
java.lang	20420	0,83	192	12455	12263	384	129,17
AVERAGE	13549,30	0,64	1616,52	1595,04	-21,48	422,30	141,48

12.5.1.3 Heuristic violations in original project

A detailed description of list of heuristic violations of the JDK class library goes beyond the scope of this document[27]. Most of the identified violations can be found in the java.awt packages. These packages are user interface components that are not the scope of this thesis.

12.5.1.4 Measurement results of transformed project

Table 34: JDK: transformed project measurements

	FragmentSize	PublicFactor	Coupling	InvCoupling	Coupling hardness	Coupling Tension	Cohesion
java.awt.datatransfer	959	0,75	335	0	-335	0	7,00
java.security	4829	0,78	2180	29	-2151	58	28,46
java.rmi.server	3325	0,75	1570	15	-1555	30	22,73
java.util	10134	0,78	991	1750	759	1982	88,75
java.awt.peer	2226	0,15	24	91	67	48	1,23
java.security.interfaces	532	1,00	2	0	-2	0	3,40
java.security.acl	903	1,00	606	0	-606	0	6,88
java	0	0,00	0	0	0	0	0,00
java.lang.reflect	3282	0,69	822	171	-651	342	0,29
java.awt.event	21350	0,93	1184	84	-1100	168	413,92
sun	0	0,00	0	0	0	0	0,00
java.io	13025	0,58	1027	2762	1735	2054	67,01
java.net	6555	0,61	1502	75	-1427	150	45,82
java.awt.image	3210	0,72	54	0	-54	0	71,93
java.rmi.registry	136	0,73	7	0	-7	0	0,33
java.rmi	4927	0,93	1552	682	-870	1364	142,79
java.sql	3185	0,38	297	0	-297	0	35,53
java.awt	90231	0,68	1086	1318	232	2172	375,89
java.applet	291	0,14	6	5	-1	10	0,67
java.math	7433	0,86	195	0	-195	0	34,33
java.util.zip	3991	0,66	848	0	-848	0	69,11
java.rmi.dgc	167	0,34	20	0	-20	0	4,67
sun.misc	1	0,00	0	1	1	0	0,00
java.beans	8474	0,57	1370	0	-1370	0	46,29
java.text	24865	0,77	1071	0	-1071	0	253,85
java.text.resources	3502	0,44	1732	0	-1732	0	0,00
java.lang	18541	0,77	194	11350	11156	388	114,78

[27] The complete list is available from the author.

	FragmentSize	PublicFactor	Coupling	InvCoupling	Coupling hardness	Coupling Tension	Cohesion
AVERAGE	8743,48	0,59	691,67	679,00	-12,67	324,67	67,99

12.5.2 Empiric Evaluation

12.5.2.1 Evaluation of heuristics: Specific comments to list of heuristic violations

In the java.awt packages the MeTHOOD heuristics have identifies many potential flaws. However, an in depth examination of the user interface components is not in the scope of this thesis.

In the java base classes, many smaller flaws can be found (e.g. duplicate attributes in different classes, redundantly defined constants, and many attributes with implementation access). Most of them can not be considered as serious, since this base library can be used in many contexts that justify nearly every heuristic violation.

12.5.2.2 Evaluation of heuristic based transformation

Table 35: JDK: difference between original and transformed

	FragmentSize	PublicFactor	Coupling	InvCoupling	Coupling hardness	Coupling Tension	Cohesion
java.awt.datatransfer	121	-0,03	26	-1	-27	-2	-6,00
java.security	-320	-0,02	-415	-23	392	-46	-3,04
java.rmi.server	158	0,01	32	0	-32	0	1,68
java.util	43	0,01	-588	-11203	-10615	-1176	-38,75
java.awt.peer	314	0,15	-11	64	75	-6	-0,21
java.security.interfaces	-34	0,00	-25	0	25	0	-0,60
java.security.acl	66	0,00	77	0	-77	0	-0,38
java	0	0,00	0	0	0	0	0,00
java.lang.reflect	62	0,00	34	-5	-39	-10	-0,14
java.awt.event	6946	0,00	-757	-65	692	-130	224,35
sun	0	0,00	0	0	0	0	0,00
java.io	546	0,02	54	-346	-400	108	-7,77
java.net	481	0,00	204	-13	-217	-26	-4,69
java.awt.image	262	-0,06	-5	-16	-11	-32	-5,00
java.rmi.registry	0	0,00	-2	-1	1	-2	-0,67
java.rmi	403	0,00	149	47	-102	94	12,00
java.sql	-263	-0,08	-419	0	419	0	6,20
java.awt	-107206	-0,17	-606	-11966	-11360	-1212	-1750,91
java.applet	-11838	-0,77	-11404	-94	11310	-188	-1,08
java.math	-777	0,06	156	-6	-162	-12	-306,67
java.util.zip	4	0,02	-45	0	45	0	-17,05
java.rmi.dgc	0	0,00	-1	0	1	0	0,00
sun.misc	0	0,00	0	0	0	0	0,00
java.beans	-280	-0,01	-365	0	365	0	1,25
java.text	4263	0,01	99	0	-99	0	40,08

	FragmentSize	PublicFactor	Coupling	InvCoupling	Coupling hardness	Coupling Tension	Cohesion
java.text.resources	-20829	-0,24	-11161	0	11161	0	-112,40
java.lang	-1879	-0,06	2	-1105	-1107	4	-14,39
AVERAGE	-4805,81	-0,04	-924,85	-916,04	8,81	-97,63	-73,49

12.6 ERSys

12.6.1 Project Assessment

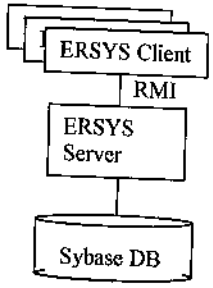


Figure 59: System architecture of ERSys

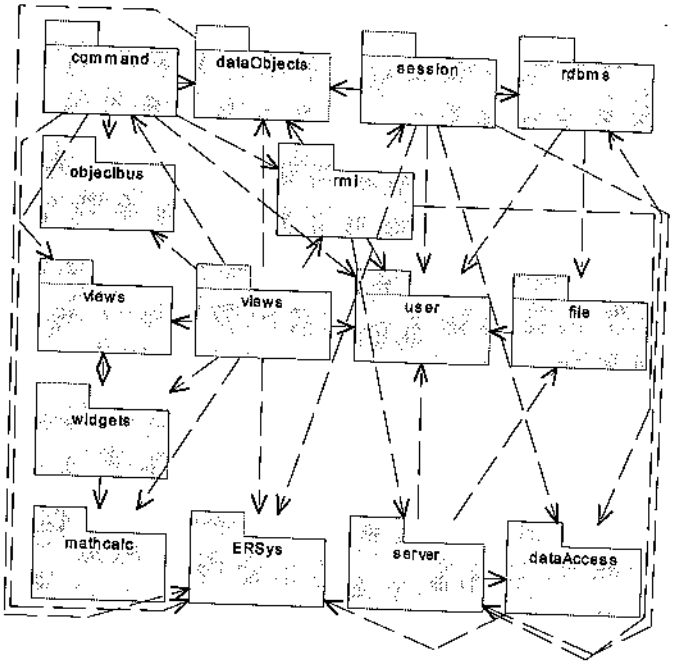


Figure 60: ERSys components

Systors ERSys (Expense Reimbursement System) (Maffeis, Toenniessen et al. 1999) is an application which allows to manage the expense reimbursement process of a large company. Employees can enter and manage their expenses which are accounted at the end of the month.

12.6.1.1 Description of the project architecture

Figure 59 illustrates the system architecture of ERSys. It consists of a Sybase database, a central ERSys server and an ERSys client, which is the end user interface of ERSys.

Figure 60 illustrates the ERSys packages and their dependencies. The packages have the following functions:

dataAccess	Container for various packages related to access of data in different data containers and exception classes which can occur during data access.
dataAccess.base	Contains different base classes and utilities for data access.
dataAccess.file	Classes for accessing and mapping data stored in files.
dataAccess.rdbms	Classes for accessing and mapping data stored in a relational database.
dataAccess.server	Base class to encapsulate data access on any kind of data server.
ERSys	Container for the ERSys core packages. Also contains the ERSysException class.
ERSys.command	Contains the implementation and infrastructure classes implementing *command objects*. A command object encapsulates a single complex action in the system, e.g., deleting an expense position.
ERSys.dataObjects	Contains the expense reimbursement business classes, e.g. *Employee* and *ExpensePosition*.
ERSys.rmi	Contains the implementation of the ERSys RMI (Remote Method Invocation) Server. This server accepts requests from ERSys clients and provides an interface for the ERSys database.
ERSys.views	Contains all ERSys user interface classes.
system	Contains business independent classes. Provides a technical framework for ERSys.
system.mathcalc	Contains an utility for controlled rounding of figures.
system.objectbus	Contains a simple bus based messaging mechanism. This package is also used in 12.3.
system.session	Contains various mechanisms needed in a user session.
system.user	Contains an abstraction of a user and various roles the user can play.
system.views	Contains various user dialogs, e.g., a splashscreen and different dialog classes.
system.widgets	Contains various common user interface utilities, e.g.,

	tool tips, base classes for text fields and icons.
test	Contains various test classes which test different aspects of the system.

12.6.1.2 Measurement results of the original project

Table 36: ERSys: original measurements

	FragmentSize	PublicFactor	Coupling	InvCoupling	Coupling hardness	Coupling Tension	Cohesion
dataAccess	32	0,79	23	0	-23	0	0,00
dataAccess.base	1711	0,88	112	366	254	224	4,08
dataAccess.file	437	0,95	30	0	-30	0	0,00
dataAccess.rdbms	696	0,89	67	0	-67	0	3,00
dataAccess.server	1035	0,99	31	88	57	62	0,50
ERSys	8	0,71	4	128	124	8	0,00
ERSys.command	3468	0,66	1025	0	-1025	0	63,31
ERSys.dataObjects	1360	0,87	521	1124	603	1042	4,67
ERSys.rmi	1228	0,94	107	509	402	214	0,80
ERSys.views	3588	0,50	1347	0	-1347	0	21,80
system	0	0,00	0	0	0	0	0,00
system.mathcalc	43	0,76	0	0	0	0	0,00
system.objectbus	106	0,73	17	0	-17	0	4,00
system.session	1300	0,91	136	55	-81	110	0,80
system.user	304	0,77	45	199	154	90	31,60
system.views	35	0,52	16	0	-16	0	0,00
system.widgets	173	0,64	45	146	101	90	0,00
test	417	0,72	90	0	-90	0	1,62
AVERAGE	885,61	0,73	200,89	145,28	-55,61	102,22	7,56

Consider the package *system.objectbus*. The content of this package is a redesigned version of the package *tors.objectbus* in 12.3. The objectbus is now used as a *singleton* (Gamma, Helm et al. 1994). This is an improvement to the use of objectbus in 12.3. If you study section 12.3.1.4, you will notice that a MeTHOOD heuristic has identified this problem in 12.3 and that in 12.3 the redesign as a singleton has been proposed. Please also note the changed values of InverseCoupling, Coupling hardness and Coupling tension of objectbus: as a singleton, objectbus is not used for associations, parameters or member variables, therefore, dependencies to objectbus are visible in the source code only, and are thus not analyzed.

12.6.1.3 Heuristic violations in original project

Table 37: ERSYS heuristic violatons

MH02	Association: [system.objectbus.ObjectBusEvent]..eventDescription Is ProtectedAccess Association: [ERSys.command.DeleteExpensePosCommand]..mExpensePositions Is ProtectedAccess Association: [ERSys.command.DeleteExpensePosCommand]..mSession Is ProtectedAccess	*Stronger Protection*

	Attribute: [ERSys.views.ViewExpensePositionThread].mCurrentReceiptNumber is ProtectedAccess Attribute: [ERSys.views.ViewExpensePositionThread].mIsChange is ProtectedAccess Attribute: [ERSys.dataObjects.ExpensePositionSet].mCurrentReceiptNo is ProtectedAccess Association: [dataAccess.base.PersistentObjectSet]..mPersistentObjects Is ProtectedAccess Association: [ERSys.command.ApproveCommand]..mAfterImages Is ProtectedAccess Association: [ERSys.command.DisapproveCommand]..mSession Is ProtectedAccess Association: [dataAccess.rdbms.RDBMSDataAccessor]..mUserName Is ProtectedAccess Association: [dataAccess.rdbms.RDBMSDataAccessor]..mPassword Is ProtectedAccess Association: [ERSys.command.Command]..mUser Is ProtectedAccess Attribute: [ERSys.dataObjects.ExpenseType].mID is ProtectedAccess Attribute: [ERSys.dataObjects.ExpenseType].mFactor is ProtectedAccess Association: [system.user.User]..mName Is ProtectedAccess Association: [system.user.User]..mLongName Is ProtectedAccess Attribute: [ERSys.command.CommandFactory].mMaxCommandsToUndo is ProtectedAccess Association: [ERSys.command.CommandFactory]..mCurrentCommandFactory Is PublicAccess Association: [system.session.DefaultSession]..mSessionID Is ProtectedAccess Association: [system.session.DefaultSession]..mUser Is ProtectedAccess Attribute: [ERSys.dataObjects.ExpensePosition].mReceiptNo is ProtectedAccess Attribute: [ERSys.dataObjects.ExpensePosition].mMonth is ProtectedAccess Attribute: [system.session.DefaultSessionManager].mCurrentUserSessionID is ProtectedAccess Attribute: [system.session.DefaultSessionManager].mTimeOutValue is ProtectedAccess	
MH14	Redundant property: [int APPROVED_STATE] occurs in: [DeleteExpensePosCommand] as [Attribute]: [ApproveCommand] as [Attribute]: [DisapproveCommand] as [Attribute]: [ChangeExpensePosCommand] as [Attribute]: [AddExpensePosCommand] as [Attribute]: [HandoverProjectAccountingCommand] as [Attribute]: Redundant property: [int UNAPPROVED_STATE] occurs in: [DeleteExpensePosCommand] as [Attribute]: [ApproveCommand] as [Attribute]: [DisapproveCommand] as [Attribute]: [ChangeExpensePosCommand] as [Attribute]: [AddExpensePosCommand] as [Attribute]: Redundant property: [int HANDOVER_STATE] occurs in: [DeleteExpensePosCommand] as [Attribute]: [ChangeExpensePosCommand] as [Attribute]: [AddExpensePosCommand] as [Attribute]: [HandoverProjectAccountingCommand] as [Attribute]: Redundant property: [String SUBJECT_COMMAND] occurs in: [DeleteExpensePosCommand] as [Role]: [ViewExpensePositionThread] as [Role]: [ApproveCommand] as [Role]: [DisapproveCommand] as [Role]: [ChangeExpensePosCommand] as [Role]: [AddExpensePosCommand] as [Role]: [ViewExpensePosition] as [Role]: [HandoverProjectAccountingCommand] as [Role]: [MainView] as [Role]: Redundant property: [ExpensePositionSet mExpensePositions] occurs in: [DeleteExpensePosCommand] as [Role]: [ApproveCommand] as [Role]: [DisapproveCommand] as [Role]: [ChangeExpensePosCommand] as [Role]: [AddExpensePosCommand] as [Role]: [HandoverProjectAccountingCommand] as [Role]: Redundant property: [String SUBJECT_NEW_EXPOS] occurs in: [ViewExpensePositionThread] as [Role]: [ViewExpensePosition] as [Role]: [MainView] as [Role]:	*Redundant constant declaration*
	Redundant property: [User mUser] occurs in: [TestDefaultDataServer] as	*Context as*

	[Role]: [Command] as [Role]: [DefaultSession] as [Role]: [MainControl] as [Role]: Redundant property: [RMISession mSession] occurs in: [DeleteExpensePosCommand] as [Role]: [ViewExpensePositionThread] as [Role]: [ApproveCommand] as [Role]: [DisapproveCommand] as [Role]: [CommandFactory] as [Role]: [ChangeExpensePosCommand] as [Role]: [AddExpensePosCommand] as [Role]: [ViewExpensePosition] as [Role]: [MainControl] as [Role]: [HandoverProjectAccountingCommand] as [Role]: [MainView] as [Role]: [DefaultCache] as [Role]: Redundant property: [String mServerName] occurs in: [RDBMSDataAccessor] as [Role]: [ERSysRMIServer] as [Role]: [RDBMSMapping] as [Role]: [RDBMSProcedure] as [Role]: Redundant property: [String mDBName] occurs in: [RDBMSDataAccessor] as [Role]: [RDBMSMapping] as [Role]: [RDBMSProcedure] as [Role]:	*parameters*
	Redundant property: [ExpensePositionSet mExpensePositions] occurs in: [DeleteExpensePosCommand] as [Role]: [ApproveCommand] as [Role]: [DisapproveCommand] as [Role]: [ChangeExpensePosCommand] as [Role]: [AddExpensePosCommand] as [Role]: [HandoverProjectAccountingCommand] as [Role]:	*Base class move*
	Redundant property: [Vector mResults] occurs in: [RDBMSDataAccessor] as [Role]: [FileDataAccessor] as [Role]: Redundant property: [Vector mResults] occurs in: [RDBMSDataAccessor] as [Role]: [FileDataAccessor] as [Role]: Redundant property: [String mValue] occurs in: [FileMappingPosition] as [Role]: [RDBMSMappingPosition] as [Role]:	*Missing base class*
	Redundant property: [String mDepartmentAccountNo] occurs in: [ExpensePositionSet] as [Role]: [Employee] as [Role]: [User] as [Role]: [MainControl] as [Role]: [Project] as [Role]: Redundant property: [String mName] occurs in: [Activity] as [Role]: [UndefinedUser] as [Role]: [Employee] as [Role]: [User] as [Role]: [Project] as [Role]: Redundant property: [String mDescription] occurs in: [Activity] as [Role]: [ExpenseType] as [Role]: [UndefinedUser] as [Role]: [User] as [Role]: [ExpensePosition] as [Role]: [Project] as [Role]:	*Missing Abstraction*
MH17	Operation: [dataAccess.base.PersistentObjectSet].getKeys occurs in all of the subclasses of [dataAccess.base.PersistentObjectSet]. It should be moved To [dataAccess.base.PersistentObjectSet] Operation: [dataAccess.base.PersistentObjectSet].getDescriptions occurs in all of the subclasses of [dataAccess.base.PersistentObjectSet]. It should be moved To [dataAccess.base.PersistentObjectSet]	*Missing base class*
MH23	[ERSys.rmi.RMISession] depends from a softer class [ERSys. dataObjects.ExpensePosition] [ERSys.rmi.RMISession] depends from a softer class [ERSys.dataObjects.ExpenseType] [ERSys.rmi.RMISession] depends from a softer class [ERSys.dataObjects.ExpenseTypeSet] [ERSys.rmi.RMISession] depends from a softer class [ERSys.dataObjects.ExpensePosition] [ERSys.rmi.RMISession] depends from a softer class [ERSys.dataObjects.ExpenseType] [ERSys.rmi.RMISession] depends from a softer class [ERSys.dataObjects.ExpenseTypeSet]	*Interface Redesign*
MH25	[system.objectbus.MainObjectBus] has a direct recursive Association	*Composite*

12.6.1.4 Measurement results of transformed project

Table 38: ERSys: transformed project measurements

	FragmentSize	PublicFactor	Coupling	InvCoupling	Coupling hardness	Coupling Tension	Cohesion
dataAccess	48	0,50	25	0	-25	0	0,00
dataAccess.base	1834	0,87	117	440	323	234	0,50
dataAccess.file	521	0,83	29	0	-29	0	0,00
dataAccess.rdbms	852	0,75	66	0	-66	0	3,00
dataAccess.server	1087	0,98	23	112	89	46	0,50
ERSys	14	0,38	4	192	188	8	0,00
ERSys.command	3533	0,68	1003	0	-1003	0	53,43
ERSys.dataObjects	1698	0,69	563	912	349	1126	4,53
ERSys.rmi	1180	0,91	98	505	407	196	0,80
ERSys.views	6002	0,53	1124	0	-1124	0	104,95
system	0	0,00	0	0	0	0	0,00
system.mathcalc	55	0,59	0	0	0	0	0,00
system.objectbus	127	0,61	17	0	-17	0	2,14
system.session	1304	0,85	125	55	-70	110	0,20
system.user	672	0,58	44	198	154	88	73,20
system.views	38	0,43	10	0	-10	0	0,00
system.widgets	233	0,47	42	105	63	84	0,00
test	484	0,61	112	0	-112	0	0,29
AVERAGE	1093,44	0,63	189,00	139,94	-49,06	105,11	13,53

12.6.2 Empiric Evaluation

12.6.2.1 Evaluation of heuristics: Specific comments to list of heuristic violations

MH02:

Stronger Protection

Most of these attributes are protected attributes. Nearly all of them can be declared as private. The weak encapsulation is not necessary. An example for this is *ERSys.command.Command*. It declares protected attributes and provides "getters" for the attributes (which are always used to access the attribute). The attributes are never used directly and it would be reasonable declaring them as private. Some attributes are public, e.g., *ERSys.command.CommandFactory.mCurrentCommandFactory* should also be declared as private.

MH14:

Redundant constant declaration

These constant declarations are redundancies. All constants with the name carry the same values. They are defined in different classes. The code depends from the fact that the values are always the same. These redundancies can be eliminated without problems.

Context as parameters

This flaw is similar to *Singleton redesign* in the corresponding section of the TOS assessment. Context information, like the current user or the current session, is provided as method parameters or member variables in all locations where they are needed. This leads to unnecessary complexity. This flaw can be removed using singletons or a registry, e.g., a registry where the current user is registered on a thread or a threadgroup.

Base class move

These properties should be moved to the existing base class. An example is the attribute *mExpensePositions* that is used (and declared) in all subclasses of *Command*.

Missing base class

In some cases, e.g. *RDBMSDataAccessor and FileDataAccessor* there are overlapping properties, but a class for these properties missing and should be added.

Missing Abstraction

These flaws consist of properties with the same name that occur in many classes of the project. Examples are *DepartmentAccountNo, Name, Description*. These properties are member variables and are used in slightly different in different contexts. Additional abstractions like *Accountable, Nameable* and *Describable* would unify these properties under a single "roof" and simplify much code.

MH17:

Missing Abstraction

This flaw is similar *Missing Abstraction* in MH14. The classes *Activy, Employee, ExpensePosition, ExpenseType* and *Project* should to implement additional small interfaces (*Describable* with a method *getDescriptions()* and *Identifiable* with a method *getKeys()*). It is not a problem to add these interfaces; the methods are already implemented on the classes. These interfaces would eliminate eight methods (most of the code) in the corresponding typed sets (*ActivitySet, EmployeeSet*, etc.), by moving the methods *getKeys()* and *getDescriptions()* to the baseclass (*PersistentObjectSet*).

MH23:

Interface Redesign

The interface RMISession depends from many classes (ExpensePosition, ExpenseType, etc.). Many other locations in the code depend from this interface. These dependencies lead to a high coupling tension of this interface and change propagation is very likely. An interface redesign is necessary. Different approaches could be used (e.g., a mediator or more generic types, especially for the sets), but the redesign is not easy to do.

MH25:

Composite

See section 12.3.2.2. *Objectbus* is a reused component in both projects. ER-Sys uses a later and improved version of this software.

12.6.2.2 Evaluation of heuristic based transformation

Table 39: ERSys: difference of original and transformed

	FragmentSize	PublicFactor	Coupling	InvCoupling	Coupling hardness	Coupling Tension	Cohesion
dataAccess	16	-0,29	2	0	-2	0	0,00
dataAccess.base	123	-0,01	5	74	69	10	-3,58
dataAccess.file	84	-0,12	-1	0	1	0	0,00
dataAccess.rdbms	156	-0,15	-1	0	1	0	0,00
dataAccess.server	52	-0,01	-8	24	32	-16	0,00
ERSys	6	-0,33	0	64	64	0	0,00
ERSys.command	65	0,02	-22	0	22	0	-9,88
ERSys.dataObjects	338	-0,17	42	-212	-254	84	-0,13
ERSys.rmi	-48	-0,02	-9	-4	5	-18	0,00
ERSys.views	2414	0,03	-223	0	223	0	83,15
system	0	0,00	0	0	0	0	0,00
system.mathcalc	12	-0,17	0	0	0	0	0,00
system.objectbus	21	-0,12	0	0	0	0	-1,86
system.session	4	-0,06	-11	0	11	0	-0,60
system.user	368	-0,18	-1	-1	0	-2	41,60
system.views	3	-0,08	-6	0	6	0	0,00
system.widgets	60	-0,17	-3	-41	-38	-6	0,00
test	67	-0,10	22	0	-22	0	-1,33
AVERAGE	207,83	-0,11	-11,89	-5,33	6,56	2,89	5,97

12.7 Customer Consultant Support System (CCSS)

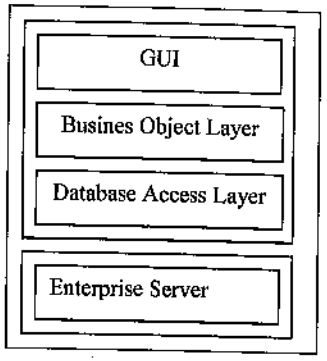


Figure 61: High level architecture of CCSS

CCSS is a system that supports private customer consultants of a bank during sales and consulting tasks. Until now, three modules of this system have been realized. The first module, called *visit planning*, supports customer consultants during preparation and completion of customer calls or visits. The second module, called *travel planing*, implements the travel allowance process of the bank. The last module called *Customer Profile* provides an extended set of customer data.

12.7.1 Project Assessment

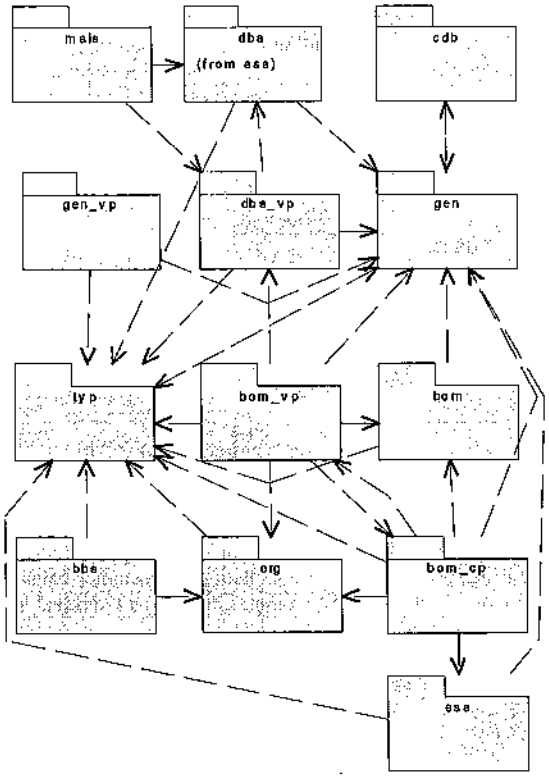


Figure 62: CCSS components

12.7.1.1 Description of the project architecture

Figure 61 shows a high level view of the architecture of the CCSS. It consists of a graphical user interface (GUI), a business object layer (BOL) and a database access layer (DAL). The modules run on Windows NT workstation and access the central enterprise server.

The GUI is the interface between users and the business knowledge in the business layer. The BOL contains the knowledge about business specific knowledge and functionality, like products, partners and business functions like reporting a visit. The DAL is the interface between the business knowledge and its persistent representation. It presents a view on the persistent data, which does not dependent from the used database management systems. Currently, it is possible to use the enterprise server, but also relational database management systems can be used via SQL.

Figure 62 illustrates the CCSS packages and their dependencies. CCVSS uses a generic framework (see Figure 63) which integrates most of the classes. The functions of most of the packages can be better explained if this framework is understood. The framework provides an infrastructure for three different kinds of classes: general or common classes (GEN) (implementing mechanism is useful for every object), database access classes (DBA)(implementing an interface to a database) and classes implementing the business object model (BOM) (containing business functionality). GENObject, DBAObject and BOMObject are the base classes for each type of class. GENObject defines mechnisms useful for all classes in the system, e.g., an objectidentifier, BOMObjects defines mechanisms for all business objects, and DBAObject defines mechanisms to read and write objects from a database. The items shown in Figure 63 play a special role. They are used to implement the primitive data members of business objects and implement the interface between DBAObjects and business objects. They know how to convert a business object in atomic data and vice versa.

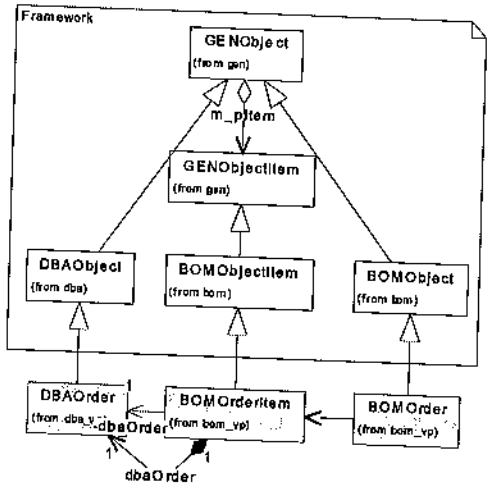


Figure 63: CCSS class framework

The packages also represent the separation between business (BOM), data (DBA) and general or common (GEN) classes The CCSS packages have the following functions:

typ	Contains different classes implementing high level data types. e.g., amount, date and string.
dba_vp	Contains database classes for visit planning business objects.
bom_vp	Contains visit planning business objects.
gen	Includes classes implementing general or common utilities used by many other classes. Also includes the implementation of general base classes (*GENObject* and *GENObjectItem*) for most other classes.
gen_vp	General classes for visit planning.
bom	Base classes and mechanisms for business objects.
cdb	Contains a *codemanager*. This is an interface to a database containing so-called *codes*. A code represents a term used in the business domain which needs to be represented in different languages and can have different descriptions (usually a long and a short description).
bom_cp	Business classes for the customer profile.
bbs	Contains an implementation of access control related features, e.g., get the current user, check permissions, etc.
esa.dba	Database classes for an interface to a host system called ESA (enterprise server access).
org	Contains an implementaion of organization related classes, e.g. ORGUnit and ORGManager.
mais	
esa	Implementation of the ESA (Enterprise Server Access) interface.
<unspecified>	Classes referenced but not defined in the schema.

12.7.1.2 Measurement results of the original project

Table 40: CCSS: original measurements

	FragmentSize	PublicFactor	Coupling	InvCoupling	Coupling hardness	Coupling Tension	Cohesion
typ	18056	0,92	2759	20316	-17557	5518	326,83
dba_vp	81283	0,90	25976	288	25688	576	1,35
bom_vp	189802	0,80	41558	0	41558	0	96,82
gen	19055	0,91	283	21691	-21408	566	31,54
gen_vp	30107	0,97	14672	0	14672	0	1,00
bom	2036	0,69	1025	23108	-22083	2050	1,57
cdb	289	0,67	48	4	44	8	27,50
bom_cp	11125	0,98	1084	0	1084	0	1,67
bbs	204	0,85	6	0	6	0	1,00
esa.dba	2145	0,75	695	29222	-28527	1390	2,40
org	1842	0,99	11	0	11	0	1,33
mais	4794	0,68	4403	0	4403	0	1,33
esa	9920	0,76	5071	0	5071	0	1,00
<unspecified>	143	0,00	1	460	-459	2	0,05
AVERAGE	26485,79	0,78	6970,86	6792,07	178,79	722,14	35,39

12.7.1.3 Heuristic violations in original project

Table 41: CCSS heuristic violations

MH02	Attribute: [bom_vp.BOMOrderItem].vrAccountArray is PublicAccess Association: [gen.GENVarException]..varType Is PublicAccess Attribute: [gen.KEYTmpKey].m_fDelete is PublicAccess Attribute: [bom_vp.BOMTaskItem].visitReportArray is PublicAccess Association: [bom_vp.BOMTripPackManager]..m_Position Is PublicAccess Association: [bom_vp.BOMTravelItem]..dbaTravel Is PublicAccess	*Stronger Protection*
MH14	Redundant property: [BOMObjectItemArray* travelArray] occurs in: [BOMVisitItem] as [Attribute]: [BOMTripPackNNPERItem] as [Attribute]: [BOMTravelDestinationItem] as [Attribute]: [BOMTravelReportItem] as [Attribute]: [BOMTravelSurveyItem] as [Attribute]; Redundant property: [TYPCode m_State] occurs in: [BOMVisitItem] as [Role]: [BOMTravelReportItem] as [Role]: [BOMTripPackItem] as [Role]: [BOMTaskItem] as [Role]: [BOMTravelSurveyItem] as [Role]: [BOMVisitReportItem] as [Role]: [BOMTravelItem] as [Role]; Redundant property: [TYPDate m_StateDate] occurs in: [BOMVisitItem] as [Role]: [BOMTravelReportItem] as [Role]: [BOMTripPackItem] as [Role]: [BOMTaskItem] as [Role]: [BOMTravelSurveyItem] as [Role]: [BOMVisitReportItem] as [Role]: [BOMTravelItem] as [Role]; Redundant property: [TYPTime m_StateTime] occurs in: [BOMVisitItem] as [Role]: [BOMTravelReportItem] as [Role]: [BOMTripPackItem] as [Role]: [BOMTaskItem] as [Role]: [BOMTravelSurveyItem] as [Role]: [BOMVisitReportItem] as [Role]: [BOMTravelItem] as [Role]; Redundant property: [TYPNString m_StateNNPER] occurs in: [BOMVisitItem] as [Role]: [BOMTravelReportItem] as [Role]: [BOMTripPackItem] as [Role]: [BOMTaskItem] as [Role]: [BOMTravelSurveyItem] as [Role]: [BOMVisitReportItem] as [Role]: [BOMTravelItem] as [Role]; Redundant property: [TYPCode m_PartnerCSITZ] occurs in: [BOMVisitItem] as [Role]: [BOMVrAccountItem] as [Role]: [BOMVrDepositItem] as [Role]: [BOMTaskItem] as [Role]: [BOMVisitReportItem] as [Role]; Redundant property: [TYPNString m_PartnerNSTAM] occurs in: [BOMVisitItem] as [Role]: [BOMTaskItem] as [Role]: [BOMVisitReportItem] as [Role]; Redundant property: [TYPCode m_PartnerCRMM] occurs in: [BOMVisitItem] as [Role]: [BOMTravelDestinationItem] as [Role]: [BOMTravelReportItem] as [Role]: [BOMTravelSurveyItem] as [Role]: [BOMVisitReportItem] as [Role]; Redundant property: [TYPCode m_PartnerCCRT] occurs in: [BOMVisitItem] as [Role]: [BOMTravelDestinationItem] as [Role]: [BOMTravelReportItem] as [Role]: [BOMTravelSurveyItem] as [Role]: [BOMVisitReportItem] as [Role]; Redundant property: [TYPCode m_PartnerCDOM] occurs in: [BOMVisitItem] as [Role]: [BOMTravelDestinationItem] as [Role]: [BOMTravelReportItem] as [Role]: [BOMTravelSurveyItem] as [Role]: [BOMVisitReportItem] as [Role]; Redundant property: [TYPCode m_PartnerCGREG] occurs in: [BOMVisitItem] as [Role]: [BOMTravelReportItem] as [Role]: [BOMTravelSurveyItem] as [Role]: [BOMVisitReportItem] as [Role]; Redundant property: [TYPNString m_ConsultantNSBCO] occurs in: [BOMVisitItem] as [Role]: [BOMTripPackNNPERItem] as [Role]: [BOMTripPackNSBCOItem] as [Role]: [BOMTravelItem] as [Role]; Redundant property: [TYPNString m_ConsultantNNPER] occurs in: [BOMVisitItem] as [Role]: [BOMTripPackNNPERItem] as [Role]: [BOMTravelItem] as [Role]; Redundant property: [TYPObjectID m_VisitID] occurs in: [BOMVisitItem] as [Role]: [BOMVisitCompanionItem] as [Role]: [BOMVisitReportItem] as [Role]; Redundant property: [TYPObjectID m_TravelID] occurs in: [BOMVisitItem] as [Role]: [BOMTravelCompanionItem] as [Role]: [BOMTravelPurposeItem] as [Role]: [BOMTravelDestinationItem] as [Role]: [BOMTravelReportItem] as [Role]: [BOMTravelSurveyItem] as [Role]: [BOMTravelItem] as [Role]; Redundant property: [TYPCode m_CompanionCSITZ] occurs in:	*Missing Abstraction/ Missing base class*

	[BOMVisitCompanionItem] as [Role]: [BOMTravelCompanionItem] as [Role]: Redundant property: [TYPNString m_CompanionNSBCO] occurs in: [BOMVisitCompanionItem] as [Role]: [BOMTravelCompanionItem] as [Role]: Redundant property: [TYPNString m_CompanionNNPER] occurs in: [BOMVisitCompanionItem] as [Role]: [BOMTravelCompanionItem] as [Role]: Redundant property: [TYPCode csitz] occurs in: [KEYDeposit] as [Role]: [KEYAccount] as [Role]: [KEYPartner] as [Role]: Redundant property: [TYPNString nstam] occurs in: [KEYDeposit] as [Role]: [KEYAccount] as [Role]: [KEYPartner] as [Role]: Redundant property: [BOMObjectItemArray* vrAccountArray] occurs in: [BOMOrderItem] as [Attribute]: [BOMVisitReportItem] as [Attribute]: Redundant property: [TYPObjectID m_VisitReportID] occurs in: [BOMOrderItem] as [Role]: [BOMVrAccountItem] as [Role]: [BOMVrDepositItem] as [Role]: [BOMTaskItem] as [Role]: [BOMVisitReportItem] as [Role]: Redundant property: [TYPObjectID m_VrAccountID] occurs in: [BOMOrderItem] as [Role]: [BOMVrAccountItem] as [Role]: Redundant property: [TYPCode m_Type] occurs in: [BOMOrderItem] as [Role]: [BOMTaskItem] as [Role]: Redundant property: [enum TYPVar::TState m_lowerRangeState] occurs in: [TYPAmount] as [Attribute]: [TYPNumber] as [Attribute]: [TYPTime] as [Attribute]: [TYPDate] as [Attribute]: [TYPTstamp] as [Attribute]: Redundant property: [enum TYPVar::TState m_upperRangeState] occurs in: [TYPAmount] as [Attribute]: [TYPNumber] as [Attribute]: [TYPTime] as [Attribute]: [TYPDate] as [Attribute]: [TYPTstamp] as [Attribute]: Redundant property: [double m_value] occurs in: [TYPAmount] as [Role]: [TYPNString] as [Role]: Redundant property: [TYPObjectID m_TravelSurveyID] occurs in: [BOMTsKeySuccessFactorItem] as [Role]: [BOMTsCompetitorItem] as [Role]: [BOMTravelSurveyItem] as [Role]: Redundant property: [TYPString m_Name] occurs in: [BOMTsKeySuccessFactorItem] as [Role]: [BOMTravelDestinationItem] as [Role]: Redundant property: [TYPCode m_ConsultantCCRT] occurs in: [BOMTripPackNNPERItem] as [Role]: [BOMTripPackCCRTItem] as [Role]: [BOMTripPackNSBCOItem] as [Role]: [BOMTravelItem] as [Role]: Redundant property: [BOMObjectItemArray* tripPackCRMMArray] occurs in: [BOMTripPackCCRTItem] as [Attribute]: [BOMTravelItem] as [Attribute]: Redundant property: [BOMObjectItemArray* tripPackNNPERArray] occurs in: [BOMTripPackNSBCOItem] as [Attribute]: [BOMTravelItem] as [Attribute]: Redundant property: [TYPDate m_DateGenerated] occurs in: [BOMTravelReportItem] as [Role]: [BOMTravelSurveyItem] as [Role]: [BOMVisitReportItem] as [Role]: Redundant property: [TYPAmount m_TotalOfExAssets] occurs in: [BOMTravelReportItem] as [Role]: [BOMVisitReportItem] as [Role]: Redundant property: [TYPAmount m_ComPMAssets] occurs in: [BOMTravelReportItem] as [Role]: [BOMVisitReportItem] as [Role]: Redundant property: [TYPAmount m_ExNonPMAssets] occurs in: [BOMTravelReportItem] as [Role]: [BOMVisitReportItem] as [Role]: Redundant property: [TYPAmount m_TargetNonPMAssets] occurs in: [BOMTravelReportItem] as [Role]: [BOMVisitReportItem] as [Role]: Redundant property: [TYPAmount m_ComNonPMAssets] occurs in: [BOMTravelReportItem] as [Role]: [BOMVisitReportItem] as [Role]: Redundant property: [TYPAmount m_ExMutualFunds] occurs in: [BOMTravelReportItem] as [Role]: [BOMVisitReportItem] as [Role]: Redundant property: [TYPAmount m_TargetMutualFunds] occurs in: [BOMTravelReportItem] as [Role]: [BOMVisitReportItem] as [Role]:	

	Redundant property: [TYPAmount m_ComMutualFunds] occurs in: [BOMTravelReportItem] as [Role]: [BOMVisitReportItem] as [Role]: Redundant property: [TYPAmount m_ExTrustsAssets] occurs in: ... Redundant property: [TYPObjectID m_TripPackID] occurs in: [BOMTripPackItem] as [Role]: [BOMTravelItem] as [Role]: Redundant property: [TYPCode m_ConsultantCRMM] occurs in: [BOMTripPackItem] as [Role]: [BOMTravelItem] as [Role]: Redundant property: [TYPString m_ApprovalComment] occurs in: [BOMTripPackItem] as [Role]: [BOMTravelItem] as [Role]: Redundant property: [ADate m_minValue] occurs in: [TYPDate] as [Role]: [TYPTstamp] as [Role]: Redundant property: [ADate m_maxValue] occurs in: [TYPDate] as [Role]: [TYPTstamp] as [Role]: Redundant property: [BOMObjectItemArray* visitArray] occurs in: [BOMVisitReportItem] as [Attribute]: [BOMTravelItem] as [Attribute]:	
MH17	Operation: [esa.dba.DBAAbstractManager].Instance occurs in all of the subclasses of [esa.dba.DBAAbstractManager]. It should be moved To [esa.dba.DBAAbstractManager] Operation: [bom.BOMAbstractManager].Instance occurs in all of the subclasses of [bom.BOMAbstractManager]. It should be moved To [bom.BOMAbstractManager] Operation: [bom.BOMAbstractManager].FreeInstance occurs in all of the subclasses of [bom.BOMAbstractManager]. It should be moved To [bom.BOMAbstractManager] Operation: [bom_vp.BOMTripPackItem].SetStateApproved occurs in all of the subclasses of [bom_vp.BOMTripPackItem]. It should be moved To [bom_vp.BOMTripPackItem] Operation: [bom_vp.BOMTripPackItem].CreateTripPack occurs in all of the subclasses of [bom_vp.BOMTripPackItem]. It should be moved To [bom_vp.BOMTripPackItem] Operation: [bom_vp.BOMTripPackItem].DeleteTripPack occurs in all of the subclasses of [bom_vp.BOMTripPackItem]. It should be moved To [bom_vp.BOMTripPackItem] Operation: [bom_vp.BOMTripPack].ParentID occurs in all of the subclasses of [bom_vp.BOMTripPack]. It should be moved To [bom_vp.BOMTripPack] Operation: [bom_vp.BOMTripPack].State occurs in all of the subclasses of [bom_vp.BOMTripPack]. It should be moved To [bom_vp.BOMTripPack] Operation: [bom_vp.BOMTripPack].StateDate occurs in all of the subclasses of [bom_vp.BOMTripPack]. It should be moved To [bom_vp.BOMTripPack] Operation: [bom.BOMObjectItem].Initialize occurs in all of the subclasses of [bom.BOMObjectItem]. It should be moved To [bom.BOMObjectItem]	Sibling Redesign
MH18	Non abstract Operation: [bom.BOMAbstractManager].NotifyCreate is overladed in all of the subclasses of [bom.BOMAbstractManager]. The differences should be distilled. Non abstract Operation: [bom.BOMAbstractManager].NotifyRead is overladed in all of the subclasses of [bom.BOMAbstractManager]. The differences should be distilled. Non abstract Operation: [bom.BOMAbstractManager].NotifyUpdate is overladed in all of the subclasses of [bom.BOMAbstractManager]. The differences should be distilled. Non abstract Operation: [bom.BOMAbstractManager].NotifyDelete is overladed in all of the subclasses of [bom.BOMAbstractManager]. The differences should be distilled. Non abstract Operation: [typ.TYPCode].operator CString is overladed in all of the subclasses of [typ.TYPCode]. The differences should be distilled. Non abstract Operation: [bom.BOMObjectItem].GetPersistentData is overladed in all of the subclasses of [bom.BOMObjectItem]. The differences should be distilled. Non abstract Operation: [bom.BOMObjectItem].SetPersistentData is overladed in all of the subclasses of [bom.BOMObjectItem]. The differences should be distilled. Non abstract Operation: [bom.BOMObjectItem].Update is overladed in all of	Sibling Redesign

	the subclasses of [bom.BOMObjectItem]. The differences should be distilled. Non abstract Operation: [bom.BOMObjectItem].Delete is overladed in all of the subclasses of [bom.BOMObjectItem]. The differences should be distilled. Non abstract Operation: [bom.BOMObjectItem].Undo is overladed in all of the subclasses of [bom.BOMObjectItem]. The differences should be distilled.	

12.7.1.4 Measurement results of transformed project

Table 42: CCSS: transformed project measurements

	FragmentSize	PublicFactor	Coupling	InvCoupling	Coupling hardness	Coupling Tension	Cohesion
typ	20594	0,84	2726	21881	19155	5452	368,42
dba_vp	85003	0,89	26584	288	-26296	576	1,02
bom_vp	205007	0,77	44373	0	-44373	0	170,73
gen	20201	0,89	259	27470	27211	518	31,46
gen_vp	36687	0,97	19383	0	-19383	0	1,00
bom	2178	0,64	1040	25289	24249	2080	1,00
cdb	376	0,52	50	4	-46	8	39,00
bom_cp	12075	0,97	1174	0	-1174	0	1,67
bbs	228	0,86	5	0	-5	0	1,00
esa.dba	2250	0,72	715	30515	29800	1430	1,40
org	1986	0,99	4	0	-4	0	1,33
mais	4989	0,66	4529	0	-4529	0	1,33
esa	11865	0,77	6302	0	-6302	0	1,00
<unspecified>	143	0,00	1	418	417	2	0,04
AVERAGE	28827,29	0,75	7653,21	7561,79	-91,43	719,00	44,31

12.7.2 Empiric Evaluation

12.7.2.1 Evaluation of heuristics: Specific comments to the list of heuristic violations

MH06:

Stronger Protection

These classes should use get- and setters. The weak encapsulation is not necessary.

MH14:

The illustrated flaws have all the same pattern: a set of sibling classes define properties with the same name and type. Examples of these are the siblings *BOMVisitItem, BOMTravelReportItem, BOMTravelSurveyItem*, etc.

Missing Abstraction

In some cases, this problem could be solved by the introduction of additional classes that would bundle some of these properties, e.g., a class *State* for the properties *m_State, m_StateDate and m_StateNNPER* and the class *Partner* for the code properties *m_PartnerCSITZ, m_PartnerCRMM, m_PartnerCCRT*, etc.

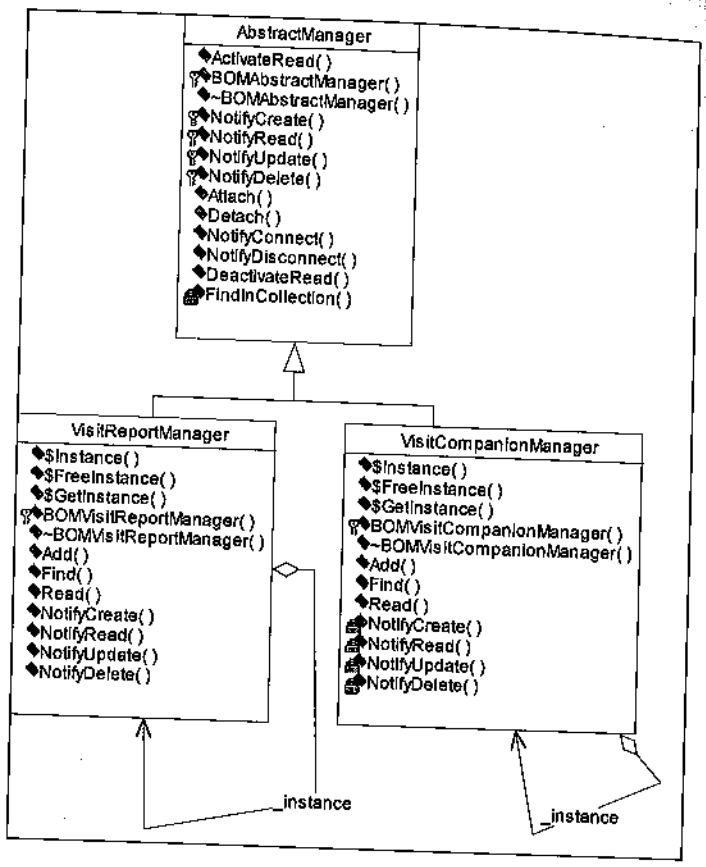


Figure 64: Overloaded operations

Missing base class

In other cases, the introduction of additional base classes would solve the problem.

MH17, MH18:

Sibling Redesign

The heuristics "Avoid full parallel overloading in siblings" and "The overloading should only define differences to the overloaded operation" are violated in many classes. One example of such a violation is the manager class hierarchy.

Figure 64 shows a section from the manager class hierarchy. The complete hierarchy has 16 classes that have nearly the same operations. The implementations of operations with the same name have an identical structure. A change in the structure of one of these operations has to be performed in 16 different source files. We have examined the source changes required to transfer the implementation to the base class. In most cases, the names of classes in the source code have to be changed from the derived class to the baseclass. For creation of objects, one additional operation is needed in the derived class. This means that the redesign (of the manager hierarchy) is feasible. It results in the elimination of most of the operations with the same name (112 operations in the model) and a reduction of about 32 pages of source code.

12.7.2.2 Evaluation of heuristic based transformation

Table 43: CCSS: difference of original and transformed

	FragmentSize	PublicFactor	Coupling	InvCoupling	Coupling hardness	Coupling Tension	Cohesion
typ	2538	-0,08	-33	1565	1598	-66	41,58
dba_vp	3720	-0,01	608	0	-608	0	-0,33
bom_vp	15205	-0,03	2815	0	-2815	0	73,91
gen	1146	-0,02	-24	5779	5803	-48	-0,08
gen_vp	6580	0,00	4711	0	-4711	0	0,00
bom	142	-0,04	15	2181	2166	30	-0,57
cdb	87	-0,16	2	0	-2	0	11,50
bom_cp	950	-0,01	90	0	-90	0	0,00
bbs	24	0,02	-1	0	1	0	0,00
esa.dba	105	-0,03	20	1293	1273	40	-1,00
org	144	0,00	-7	0	7	0	0,00
mais	195	-0,02	126	0	-126	0	0,00
esa	1945	0,01	1231	0	-1231	0	0,00
<unspecified>	0	0,00	0	-42	-42	0	-0,01
AVERAGE	2341,50	-0,03	682,36	769,71	87,36	-3,14	8,93

12.8 Summarized Evaluation of all Projects

This section gives a summary of the results of the project evaluation. It contains the results of the measurement, heuristic and transformation rule evaluation.

12.8.1 Evaluation of measures

Table 44 shows the correlation coefficients between measured and estimated values of the projects of which authors were able to provide estimations. If we take the explained (strong) variations of single properties

in the iBus project into account (see figures in parentheses), these figures are a first strong hint that the proposed measures express what developers expect from them, and that the measures are valid.

Table 44: Variation between estimated and measured values

	Fragment Size	Public Factor	Coupling	Inverse Coupling	Coupling hardness	Coupling Tension	Cohesion
iBus	0,916 (0,995)	0,880 (0,880)	0,303 (0,677)	0,607 (0,997)	0,983 (0,983)	0,983 (0,968)	0,999 (0,999)
TOS	0,993	0,958	0,982	0,978	0,926	0,98527	0,959

Furthermore, the evaluation shows that we can make valid and interesting statements about the component without a close examination. We are able to identify large and small components, and we can make valid statements about the roles they play in the systems, using the dependency measures. Most important, we are able to *compare* properties of different *component variations* that result from heuristic based transformation.

The evaluation demonstrated that the measurements can be automatically collected (even on large realistic schemas), and that they can be reproduced. Besides this, that the measures are *sensitive to small changes*. E.g., the size measure "recognizes" each change that can be performed on a design fragment, taking inheritance, composition and encapsulation into account.

12.8.2 Evaluation of heuristics

The project evaluation demonstrates that the proposed heuristics are useful for finding design flaws in a given project. The proposed heuristics found many *potential* design flaws in the examined schemas (see Table 45).

Table 45: Potential design flaws

	TOS	iBus	JDK	ERSys	CCSS	FM[28]
MH02	3	60	792	467	119	2659
MH06	229	12	132	3	1	34
MH08	0	2	7	0	0	7
MH11	490	138	864	459	547	2659
MH14	50	25	205	122	67	275
MH15	0	0	7	0	0	0
MH17	7	0	37	25	119	17
MH18	0	0	11	0	11	0
MH23	10	31	375	47	47	1097
MH24	20	19	296	46	117	13
MH25	2	2	0	4	0	19
SUM	811	289	2726	1173	1028	6793

The potential design flaws have been evaluated. The result of the evaluation is that many small *actual* design flaws could be identified in every project. In all projects, except iBus, the proposed heuristics also identified *larger*

[28] FM (financial model) is another examined project, the details of which cannot be published here.

design flaws, where a major redesign of parts of the system or central concepts can improve the quality of the systems. Examples for those locations are the missing singletons in TOS and the massive code copies in CCSS. *Therefore, the evaluation strongly supports the claim that the heuristics in the proposed approach can successfully be used to improve a design.*

The evaluation also shows an important property of the implementation of the *checking rules*. The granularity of some of the checking rules is *too coarse*. This means that the rule detects too many *potential* design flaws that are no *actual* design flaws. Especially the checking rules of the heuristics MH2, MH11 and MH23 still need refinement in this direction. MH2 signals conflicts if *public* constants do not follow a complete standard[29] (e.g. *public final static String CONSTANT="CONSTANT"*). The definition of this naming standard can be relaxed in different ways to avoid too many conflicts with MH2. MH11 also considers associations as weak that are not defined as *containment*. Only a few models include this level of information, and the used Java reverse engineering tools do not recognize contained classes. Removing the containment restriction would result in fewer conflicts with MH11. MH23 also considers classes in different class hierarchies, where the class hierarchies are orthogonal to each other, e.g., a class hierarchy of exceptions and a class hierarchy of business classes. It could be refined by considering only subtrees of a common hierarchy.

12.8.3 Evaluation of transformation rules

Table 46 shows the differences of the average measured values in the original and the transformed projects.

Table 46: All: original and transformed differences

Difference Packages	Fragment Size	Public Factor	Coupling	Inverse Coupling	Coupling hardness	Tension	Cohesion
TOS	79,80	-0,03	-56,13	-37,47	18,67	-0,40	2,04
lBus	-41,83	-0,03	-286,33	-284,67	1,67	5,67	19,89
Jdk 1.1.4	-4805,81	-0,04	-924,85	-916,04	8,81	-97,63	-73,49
CCSC	2341,50	-0,03	682,36	769,71	87,36	-3,14	8,93
FM	47783.16	-0.14	-847	-934.5	-87.5	-880.33	125.63
ERSys	207,83	-0,11	-11,89	-5,33	6,56	2,89	5,97

These figures give an overview of the effects of the default implementations of the transformation rules on the project. The *existence* of these figures closes the circle (measurement->application of heuristics->heuristic based transformation->measurement) defined in the MeTHOOD process and demonstrates that it is possible to apply this process.
The figures can be interpreted as follows: the default transformations

1. improve the Public factor,
2. reduce coupling and inverse coupling,

[29] often used in Java applications

3. raise coupling hardness,
4. improve coupling tension, and
5. lead to stronger cohesion

in most of the projects. This demonstrates that it is possible to perform a set of transformations automatically. However, we have observed that in many cases the applied transformations are not optimal. A designer would have selected another transformation for many detected problems. This shows that a completely automatic transformation is possible only in very simple cases. This observation confirms our statement that a full automatic design check and improvement is not useful. Furthermore, these results suggest that a semi-automatic process (like the proposed MeTHOOD process) is needed and feasible.

12.8.4 Evaluation of MeTHOOD integration and process

The evaluation demonstrates that

1. it is useful to represent design knowledge as proposed by the MeTHOOD integration schema,
2. this representation enables an iterative design improvement process which is based on design knowledge.
3. this process is semi-automatic, and it is not useful to automate it completely.

13 Summary, Conclusions

13.1 Summary

We have described MeTHOOD, a new extension for object-oriented design methodologies. Our approach includes

- a set of well-defined measures,
- a set of heuristics with corresponding transformation patterns,
- an integration framework for measures, heuristics and transformation rules,
- a design tool.

The core of this extension is the MeTHOOD integration schema and the MeTHOOD process. The MeTHOOD integration schema is used to integrate measures, heuristics and transformation rules. The MeTHOOD integration schema is the foundation for the MeTHOOD process. This process supports designers in continuously monitoring their conceptual design schemas, checking their conceptual design schema for design flaws and transforming flawed specifications.

Besides the MeTHOOD integration schema and process, we have also described the architecture and prototypical implementation of the MEx system. MEx is integrated in a commercially available design tool. It supports the design monitoring, checking and transformation activities of the MeTHOOD process. MEx is based on a continuously growing design knowledge catalogue containing measures, heuristics and transformation rules.

We have validated MeTHOOD based on design reviews with real applications. All of them include an object-oriented class schema that has to be shared by some other applications.

13.2 Main contributions

In this thesis, we have *demonstrated the need for a better representation of design knowledge* in order to improve the quality of object-oriented schemas. We have proposed the MeTHOOD integration schema as *a new design knowledge representation technique*. The benefits of this integration schema are the following: It provides effective support for design refinement and improvement. It reduces one of the most important weaknesses of pure measures, their passiveness. It provides concrete design advice, helping to improve the value of a measure. It solves two major problems of pure design heuristics: It provides advice on how a design fragment can be transformed if a design heuristic is violated, and gives a clear base for decisions if two design heuristics with different interests collide.

We have shown how this representation can be used as a base for *methodology extension* for existing object-oriented design methodologies by providing a *design process extension – the MeTHOOD process*. This process extension takes advantage of the design knowledge and can be used to improve design schemas iteratively and systematically.

The applicability of the integration schema has been demonstrated in the area of objectbase design. We have shown that in this area more methodology support is needed, and how scattered design knowledge can be provided, by *integrating three catalogues for objectbase design*, containing measures, heuristics and transformation rules.

We have evaluated the executability of design knowledge specified in the MeTHOOD integration by

- prototypical implementing and
- applying on a set of realistic applications

an integrated *measurement, heuristic checking and schema transformation tool* called MEx.

During our work with MeTHOOD, we made the following observations:

1. It is possible to make scattered object-oriented design knowledge tangible for designers in an effective way.
2. It seems to be impossible to describe schema quality as a whole (this observation is also supported by others (Boehm 1978; Zuse 1995; Kitchenham, Pfleeger et al. 1996)).
3. There are different approaches to describe non-trivial aspects of schema quality. With non-trivial, we mean that the set of "high quality schemas" is not restricted to a single possible structure or a hierarchy of structures like in algorithmic based relational database design (Yao 1985; Teorey, Yang et al. 1986). One approach is to measure internal properties of schema elements (e.g. the ratio between public and private attributes)

assessing aspects of *internal product quality*. Another approach is to measure conflicts to a set of heuristics (*heuristic quality*).

4. It is possible to recognize all locations in all schemas that can be changed to improve heuristic quality for a specific set of heuristics. It is possible to recognize some locations in some schemas that can be changed to improve the internal product quality. Some heuristics are related to some measures: if conflicts in these heuristics are removed, measured values are improved.
5. For some detected locations, it is possible to generate alternative improved schemas.
6. Location recognition and generation of alternative schemas can be automated.
7. Some characteristics of applications have an influence on the relevance of heuristics and measures. E.g., the lifetime of certain applications in the area of security trading is very short. Here, heuristics and measures considering aspects of maintenance are unimportant.

Using the MeTHOOD process, designers

- detect more potential design flaws early in the design phase and
- base design decisions on a clear rationale.

In summary, the *main contribution of this thesis* is a knowledge representation technique and a design knowledge base for objectbase design. It can be used to extend *existing* methodologies and enables

- a systematic design quality improvement process and
- tool support based on design knowledge.

It allows designers to

- specify explicitly desired quality parameters and better understand their interrelationships
- improve objectbase designs according to well-defined guidelines and patterns and
- use a design support tool which proposes design alternatives, transforms and documents a design.

We expect that the association of quality parameters to the design model will base object-oriented designs on better foundations, thus improving their quality.

Finally, the usage of MEx and MeTHOOD should lead to

- a higher learning curve for object-oriented designers,
- reduced costs and effects of design reviews and
- better quality of resulting software in terms of detectable design flaws.

This should lead to better reusability, extensibility and maintenance of the resulting software. Furthermore, we claim that the overhead and the level of experience, necessary to build quality software, will be lowered by systematic design transformations proposed by the design tool.

Appendix A: Design support in tools

ENVY/QA R1.0

Envy/QA is a suite of quality assurance tools for Smalltalk programs. This suite consists of tools for code reviews (code critic), metrics assessment (code metrics), regression testing tools (code coverage) and publishing (code publisher) and formatting tools (code formatter). Then tools interesting for the MeTHOOD approach are *code metrics* and *code critic*.

Code Metrics

Code Metrics provides an extensible set of Smalltalk code metrics. Such a metric has an upper and a lower threshold. Code Metrics computes the results for the specified metrics and notifies the designer if results are larger than the upper threshold or lower than the lower threshold. The metrics are categorized into:

1. Application metrics
2. Number of classes in an application + subapplications
3. Number of dependent applications
4. All prerequisites
5. Number of extended classes in an application
6. Memory size
7. Memory size for applications
8. Number of immediate prerequisites of an application
9. Class metrics
10. Number of accessor operations of a class
11. Number of class operations of a class and its baseclasses
12. Number of object operations
13. Number of object variables
14. Number of subclasses of a class
15. Number of classes coupled with a class. Class A is coupled with class B if A calls operations implemented in B.
16. Number of class operations of a class
17. Class response: number of messages sent by operations of a class
18. Number of variables of a class
19. Cyclomatic complexity of a class: Sum of exit points for all operations of a class
20. Depth of a class in the class hierarchy
21. Number of operations that reference an object, class or object variable directly (without accessor).
22. Number of references to pool or global variables from the operations of a class
23. Number of object operations
24. Memory size of a class
25. Number of new operations of a class
26. Number of pool dictionaries
27. Ratio of public to private operations

28. Number of refined operations
29. Specialization index: refined object operations * base classes/ all object operations
30. Operation Metrics
31. Lines of code of a method
32. Memory size for operations
33. Operation density: Number of statements/lines of code
34. Number of statements

For every metric, advice is given that indicates what large and small values might mean. Furthermore, for every metric, upper and lower thresholds are given. The ENVY/QA R1.0 framework allows to write project specific metrics.

Code Critic

Code Critic is a system which checks Smalltalk code for a specific set of problems. It is used to warn developers that there might be a problem in the code. In ENVY/QA, this set of problems is described by a set of so-called "Reviews". A review is a specific type of measure performed over code elements. In MeTHOOD terminology, a Review is a code heuristic. There are three categories of reviews: Application reviews, class reviews and operation reviews:

1. Application reviews
2. Missing WasRemovedCode: application performs action if it is loaded, but has no actions if it is unloaded
3. Class Reviews
4. Duplicate pool dictionaries
5. Extends base class: generic base classes should not be extended
6. Missing class comment
7. Missing dependent method
8. Not categorized operations
9. Poorly named state variables
10. Subclass responsibility: if a class does not override a necessary method
11. Subclasses base class: base classes should not be subclassed
12. Unreferenced class
13. Unused pool dictionaries
14. Unused state variables
15. Operation reviews
16. Compiler warnings
17. Could be cascaded: path of messages
18. Could use self: uses base class operation if subclass operation is not defined
19. Defeats compiler optimization
20. Direct state variable access
21. Identical to inherited method
22. Inefficient convenience method: compiler optimization
23. Magic values
24. Missing yourself

25. Missing operation comment
26. Missing primitive fail code
27. Not implemented in base class
28. Poorly named method
29. Poorly named variables
30. Public private inconsistency in sub baseclass
31. References development classes
32. References global variables
33. References outside prerequisites
34. References own class
35. Reimplements system method
36. Sends system method
37. Sent but not implemented
38. Should call baseclass
39. Should not be implemented
40. Should use isEmpty
41. Too many consecutive concatenations
42. Too many consecutive messages
43. Unnecessary #isNil #notNil
44. Unnecessary parentheses
45. Unsent method
46. Unused arguments

All reviews have a severity level that indicates how severe the problem detected by the review is. It is possible to extend Code critics by project specific reviews.

McCabe Tools

McCabe tools provide support for collecting polymorphism, inheritance, quality and encapsulation metrics.

SOMATiK

SoMATIK is a modeling tool that supports the SOMA (Graham 1995) methodology. It is mainly focused on the analysis part (especially for RAD Workshops proposed by (Graham 1995)) but is also usable for design and development. Two features of special interest for designers are *testing and execution* of specifications and the strong support for *project estimation metrics*. The testing and execution facilities allow users to execute and simulate a specified object model. In the model users can specify event traces that can be simulated later. During the simulation, new objects can be created and users can enter attribute values. The simulation facility is mainly used to test an object model with users of the system (simulation of a walkthrough). However, it can also be used to inspect the model at the design stage. The simulation and execution of specifications is a very important design support feature which is currently only supported by a few design tools. It provides designers a clear view on the dynamic behavior of the modeled system, which is nearly impossible with "static specifications" of

dynamic behavior. The SoMATIK metrics support covers many object-oriented metrics needed for project estimation (Table 47).

Table 47: SoMATIK metrics

Summary Metrics		Task Metrics	Class Metrics		Classification
	Task Statistics Summary	**Statistics**	**Summary of the Class Statistics**	**Fan-in for Classification**	**Class Classification Depth**
Number of External Entities	Number of Tasks	Fan-in for Classification	Number of Classes	Fan-in for Composition	Class Classification Width
Number of Messages	Number of Atomic Tasks	Fan-in for Composition	Number of Isolated Classes	Fan-in for Association	Class Composition Depth
Number of Actors	Number of Isolated Tasks	Fan-in for Association	Number of Classification Root Classes	Fan-in for Usage	Class Composition Width
Number of Tasks	Number of Classification Root Tasks	Fan-in for Exceptions	Number of Composition Root Classes	Fan-out for Classification	Task Classification Depth
Number of Atomic Tasks	Number of Composition Root Tasks	Fan-out for Classification		Fan-out for Composition	Task Classification Width
Number of Classes		Fan-out for Composition		Fan-out for Association	Task Composition Depth
		Fan-out for Association		Fan-out for Usage	Task Composition Width
		Fan-out for Exceptions		Number of Operations	
		Mean Complexity of RuleSets		Mean Complexity of Operations	
		Complexity of the Task Script		Number of RuleSets	
		Number of RuleSets		Number of Attributes	
				Number of Windows	
				Number of Class Messages	
				Mean Complexity of RuleSets	

The provided metrics can be categorized in three categories:

1. Schema element count,
2. Fan-in and fan-out metrics and
3. Complexity metrics.

Appendix B: Glossary

SOFTWARE QUALITY MODEL

A software quality model is a statistical relationship between a product and different ways of use of the product, its external attributes, its internal attributes and proposed measures.

MEASUREMENT

Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules. (Fenton, Whitty et al. 1995)

ATTRIBUTE

An attribute is a feature or property of the entity we are interested in. We measure attributes of entities.

INTERNAL ATTRIBUTE

An internal attribute can be measured purely in terms of the entity itself.

EXTERNAL ATTRIBUTE

An external attribute can only be measured with respect to how the entity relates to its environment.

MEASURE

A measure is an empirically objective assignment of a number (or symbol) to an entity to characterize a specific attribute. A measure is not a number but a defined mapping for the entities and attributes in question. (Fenton, Whitty et al. 1995)

DIRECT MEASUREMENT

Measurement which does not depend on the measurement of any other attribute.

INDIRECT MEASUREMENT

Measurement involving measurement of any other attribute.

DESIGN RATIONALE

A design rationale explains why an artifact is designed the way it is; it primarily justifies design decisions.

OBJECT-ORIENTED DESIGN

Process of analyzing the model produced during analysis; optimizing it to reduce coupling, increase cohesion, increase polymorphism and reuse redundancy, factoring out common behavior in a class hierarchy.

SOFTWARE SYSTEM

A system is controlled by a project team and fulfils a well-defined purpose. An example of a software system is a stock trading system. Each system consists of multiple *components*.

COMPONENTS

Components are the "integral" parts of the architecture of a system, separated by domain specific or technical boundaries. The domain specific separation, for example between products and customers, often leads to a separation between the customer and the product databases. Examples of technical boundaries are processes, operating systems and tools selected for the development of a component.

OBJECTBASE

An objectbase is a logically coherent collection of objects. An objectbase contains the state (data) *and* the behavior of objects. An objectbase is a software component, which receives messages from other components. It has always an object-oriented interface specification describing which messages are understood. One of the most important aspects of objectbases is that they are shared (used at the same time) by other components. Sharing of an objectbase can be viewed as a special case of reuse ("reuse at runtime"). A database schema, for example, is the interface specification of a shared database. A header file is often the interface specification shared of a shared program, etc.

DESIGN FLAW

A design flaw can be embodied in a structure of a design schema. It leads to non desirable properties of the schema. An example of a design flaw is an occurrence of an attribute representing the same entity (e.g., an address) in two different classes (e.g., customer and contract), which can lead to update anomalies.

QUALITY PARAMETER

A quality parameter of a component is a desirable property that can be measured. Different design solutions for the same problem often have varying quality parameters. One is more flexible, one is less complex, one is more flexible but also more complex. The designer has to find the solution that fits best to the requirements. Quality parameters describe these requirements.

A large variety of such quality parameters exists, e.g.,

1. simplicity of design,
2. understandability of the schema,
3. ease of implementation with given construction environment and
4. consistency in design.

A methodology cannot order these parameters according to importance; however, for a given problem, a designer can weigh them. The weighed parameters can be used to find the best solution (a good design), to make his decisions more explicit and transparent and to document the solutions.

Well-defined quality parameters are the basis for the transformation. They allow the designer to find discrepancies between the requirements and the existing design.

HEURISTIC

A heuristic is a rule of thumb. It is an advice on how to use design techniques in order to solve design problems. It provides guidelines for finding appropriate solutions.

Examples of general heuristics are:

- Minimize the number of classes a class collaborates with.
- Keep related data and behavior in one place.
- Distribute system intelligence uniformly.

A designer can use *heuristics* to guide himself in the right direction and trade-off between different possible solutions.

PATTERN

A pattern describes a solution to a design problem in a specific context. A pattern is defined as "a solution to a problem in a context". The description of a pattern often includes its context, a variety of typical examples, reasons for applying the pattern and a structure describing the solution. Patterns and heuristics have much in common (Riel 1995). Certain heuristics can be expressed as patterns, while others cannot. Heuristics do not contain context information, which makes them more general, broadly applicable, but at the same time harder to use. A heuristic is only a general advice that can be used in different contexts, and the designer has to provide a concrete solution for his problem.

There may be "solution patterns" which illustrate how to implement the solution, and "problem patterns" which represent the problem in the schema (e.g., a "time bomb": a class where changes lead to a reorganization of the entire system).

TRANSFORMATION RULE

A transformation rule (consider (Riel 1995) for the similar idea of transformation pattern) consists of a problem pattern and a solution pattern.

DESIGN

In general, design is an activity for describing *how* a system should work. Examples of systems are a software system (software design) or a business system, such as a set of retail business processes (business design).

ANALYSIS

Analysis is an activity that investigates *what* must be performed by the system. Analysis and design are performed iteratively, and the results of the analysis are often the input for design. Analysis and design can be considered as object-oriented if object-oriented techniques and description means (based on an object model) are used.

SOFTWARE METHODOLOGY

A full software methodology includes:

- a set of consistent concepts and models/fundamental modeling concepts,
- a set of views and notations, ideally supported by tools,
- a full life cycle process (step-by-step, iterative),
- collections of hints, rules of thumb or guidelines capturing experience, best practice and best current theory,
- a full description of deliverables,
- a set of metrics, advice on quality, test strategies,
- guidelines for project management,
- advice on library management and reuse and
- identification of organizational roles.

CONTAINMENT RELATIONSHIP

An object of a class A can contain objects of another class B. The usual containment relationship is modeled as an attribute of type B in A. This means that every object in class A contains objects of another class. This does not imply that every object of B is contained in A.

Strong Containment

The strong containment relationship reduces the visibility of the contained class definition to its container. This means that only objects of the container class can send messages to the contained objects.

MAINTAINABILITY

A software system is maintainable if it can be modified to adapt external changes, such as changes in hardware or operation systems, or to correct deficiencies and improve performance. High maintainability requires weak coupling between parts and a high level of encapsulation.

EXTENSIBILITY

A system is extensible if it can be changed to support changes towards its original purpose (add new components, capture and isolate changes to existing components).

REUSABILITY

A system is reusable if its parts or its design can be utilized to build a new system. Each component is designed around a single purpose.

References

Abadi, M., Cardelli, L. (1996). A Theory of Objects. New York, Springer.

Abbott, D. H., Korson, T. D., et al. (1994). A proposed design complexity measure for object-oriented development. Technical Report TR 94-105, Clemson University.

Abreu, F. B., Carapuca, R. (1994). Candidate Metrics for Object-Oriented Software within a Taxonomy Framework. Journal of Systems and Software. **26:** 87-96.

Abreu, F. B., Melo, W. (1996). Evaluating the Impact of Object-Oriented Design on Software Quality. 3rd Int'l Software Metrics Symposium, Berlin, Germany.

Alexander, C. (1964). Notes on Synthesis of Form. Cambridge, MA, Harvard University Press.

Alexander, C. (1979). The Timeless Way of Building. New York, Oxford University Press.

Andonoff, E., Salaberry, C., et al. (1992). Interactive Design of Object Oriented Databases. CAiSE Conf. 1992.

Andonoff, E., Salaberry, C., et al. (1992). A method for object-oriented database design. 5th International Conference on: Putting into Practice method and tools for information system design, Nantes, France.

Arnold, P., Bodoff, S., et al. (1991). An evaluation of five object-oriented development methods. Technical Report HPL-91-52. Bristol, United Kingdom, Hewlett Packard Laboratories.

Arora, R. (1997). Using Enterprise Java, Que.

Atkinson, M., Bancilhon, F., et al. (1989). The object-oriented database system Manifesto. 1st International Conference in Deductive and Object-Oriented Databases, Kyoto.

Bancilhon, F., et. al. (1992). Building an Object-Oriented Database System. The Story of O2, Morgan Kaufmann.

Barry, D. (1996). The Object Database Handbook: How to Select, Implement, and use Object-Oriented Databases. New York, Wiley.

Batini, C., Ceri, S., et al. (1992). Conceptual Database Design, Benjamin/Cummings.

Batra, D., Antony, S. R. (1994). "Novice errors in conceptual database design." European journal of Information systems **3(1)**: 57-69.

Batra, D., Zanakis, S. H. (1994). "A conceptual database design approach based on rules and heuristics." European Journal of Information Systems **3**(3): 228-239.

Beck, K., Johnson, R. (1994). Patterns Generate Architecture. ECOOP 94, Kaiserslautern.

Bernstein, P. A. (1976). "Synthesising Third Normal Form Relations from Functional Dependencies." ACM TODS **1**(4): 227-370.

Bertino, E., Marino, L. (1993). Object-oriented Database Systems, Addison Wesley.

Bieman, J. M., Kang, B.-K. (1995). Cohesion and Reuse in an Object-Oriented System. ACM Software Reusability Symp. (SRS'94).

Boehm, B. W. (1978). Characteristics of Software Quality. Amsterdam, North-Holland.

Booch, G. (1991). Object-oriented Design with applications, Bemjamin Cummings.

Booch, G. (1993). Object-Oriented Design with Applications 2nd Edition. Menlo Park, CA, Benjamin/Cummings.

Booch, G. (1995). Object solutions: Managing the object-oriented project, Addison Wesley.

Booch, G., Rumbaugh, J., et al. (1996). The Unified Modeling Language for Object-Oriented Development, Rational.

Briand, L., Morasca, S., et al. (1994). Defining and validating high-level design metrics. Technical Report CS-TR-3301, Department of Computer Science, University of Maryland.

Brodie, M. L. (1982). On the Development of Data Models. On Conceptual Object Modelling,. M. L. Brodie, J. Mylopoulos and J. W. Schmitt. New York, Springer.

Buhr, R. J. A. (1991). Practical Visual Techniques in System Design with Applications in Ada. Englewood Cliffs, NJ, Prentice Hall.

Buschmann, F., et. al. (1996). Pattern-Oriented Software Architecture - A System of Patterns, Wiley.

Cattell, R. G. G. (1994). The Object Database Standard: ODMG-93, Morgan Kaufmann Publishers, San Mateo, CA.

Ceri, S., Fraternali, P. (1997). Designing Database Applications with Objects and Rules, Addison Wesley.

Champeaux, D., Lea, D., et al. (1992). The Process of Object-Oriented Design. Conference on Object-Oriented Programming Systems, ACM Press.

Chen, P. P. S. (1976). "The Entity Relationship Model: Toward a Unified View of Data." ACM Transactions on Database Systems **1**(1): 9-36.

Chidamber, S., Kemerer, C. (1991). Towards a metric suite for object-oriented design. OOPSLA'91 Conference on Object-oriented Programming Systems, Phoenix, Arizona.

Chidamber, S. R., Kemerer, C. F. (1994). "A Metrics Suite for Object Oriented Design." IEEE Transactions of software engineering **20**(6): 476-493.

Coad, P., North, D., et al. (1995). Object Models: Strategies, Patterns, & Applications. Englewood Cliffs, Prentice Hall.

Coad, P., Yourdon, E. (1990). Object-Oriented Design. Englewood Cliffs, Yourdon Press, Prentice Hall.

Coad, P., Yourdon, E. (1992). Object-Oriented Analysis 2nd Edition, Yourdon Press, Prentice Hall.

Codd, E. F. (1972). "A Relational Model of Data for Large Shared Data Banks." Communications of the ACM **13**(6): 377-387.

Colbert, E. (1989). The object-oriented software development method: a practical approach to object-oriented development. TRI-Ada, New York.

Coleman, D., et. al. (1992). Object-Oriented Development - The Fusion Method, Prentice-Hall.

Coplien, J. O., Schmidt, D. C. (1995). Pattern Languages of Program Design, Addison Wesley.

Coplien, J. O., Schmidt, D. C. (1996). Pattern Languages of Program Design 2, Addison Wesley.

Cornell, G., Horstmann, C. (1996). Core Java, Prentice Hall.

Cunningham, W. (1995). The CHECKS Pattern language of Information Integrity. Pattern Languages of Program Design. J. O. Coplien and D. C. Schmidt, Addison Wesley.

D'Souza, D. F., Wills, A. C. (1997). Component-Based Development using Catalysis. Technical Report Draft v 0.8, ICON Computing.

Dahl, O.-J., Nygaard, K. (1966). "SIMULA - An ALGOL-Based Simulation Language." Communications of the ACM **9**(9): 671-678.

Davis, A. M., Delcambre, L. M. L. (1989). Automatic Validation of Object-Oriented Database Structure. Data Engineering 1998.

Dayal, U., Buchmann, A., et al. (1988). Rules are Objects Too: A Knowledge Model For An Active Object-Oriented Database System. 2nd Int'l. Workshop on Object-Oriented Databases, Bad Muenster, Germany.

Elmasri, R., Navathe, S. B. (1989). Fundamentals of Database Systems. Redwood City, CA, Benjamin/Cummings.

Fagin, R. (1977). Multivalued Dependencies and a New Normal Form for Relational Databases. ACM Transactions on Database Systems.

Fenton, N. (1991). Software Measurement: Why A Formal Approach? Formal Aspects of Measurement. T. Denvir, R. Herman and R. Whitty, Springer.

Fenton, N. (1991). Software Metrics: A Rigorous Approach. London, Chapman and Hall.

Fenton, N. (1994). "Software Measurement: A Necessary Scientific Basis." IEEE Transactions on Software Engineering **20**(3): 199-206.

Fenton, N., Whitty, R., et al. (1995). Software Quality Assurance and Measurement - A Worldwide Perspective, International Thomson Computer Press.

Firesmith, D., Henderson-Sellers, B., et al. (1998). Open Modeling Language (OML) Reference Manual, SIGS.

Firesmith, D. G. (1993). Object-oriented requirements analysis and logical design, Wiley.

Flanagan, D. (1996). Java in a Nutshell, O'Reilly&Associates.

Foote, B., Yoder, J. (1997). Big Ball of Mud. Pattern language of object-oriented programming, Monticello, Illinois.

Fowler, M. (1993). Position Paper on Process and Metrics for OO A&D. OOPSLA 93 Metrics Workshop.

Fowler, M. (1996). Analysis Patterns, Addison Wesley.

Frakes, W., Terry, C. (1996). "Software Reuse: Metrics and Models." ACM Computing Surveys **28**(2): 415-435.

Gamma, E., Helm, R., et al. (1994). Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley.

Gatziu, S., Dittrich, K. (1993). Events in an Active Object-Oriented Database System. 1st Intl. Workshop on Rules in Database Systems, Edinburg.

Gibbon, C., Higgins, C. (1996). "Teaching Object-Oriented Design with Heuristics." SIGPlan Notices **31**(7): 12-16.

Gorman, K., Cgoobineh, J. (1991). An overview of the object-oriented entity-relationship model (OOERM). Twenty-Third Annual Hawaii International Conference on System Sciences, Hawaii.

Gosling, J., Joy, B., et al. (1996). The Java Language Specification, Addison Wesley.

Gosling, J., Joy, B., et al. (2000). The Java Language Specification, Addison Wesley.

Graham, I. (1993). Object-Oriented Methods. Workingham UK, Addison Wesley.

Graham, I. (1995). Migrating to Object Technology. Workingham UK, Addison Wesley.

Graham, I., Henderson-Sellers, B., et al. (1997). The Open Process Specification, Addison Wesley.

Harel, D. (1987). "Statecharts: A visual formalism for complex systems." Science of Computer Programming **8**(3): 231-274.

Harrison, R. (1993). Assessing the Quality of Object-Oriented Programs. OOPSLA 93 Metrics Workshop.

Hastings, T. E. (1996). Software Size Measurement Foundations. Technical Report H96. Caulfield, Department of Software Development, Monash University.

Hay, D. C. (1995). Data Model Patterns: Convention of Thought, Dorset House Publishing.

Henderson-Sellers, B. (1995). "Who needs an object-oriented methodology anyway?" Journal of object-oriented programming **8**(6): 6-8.

Henderson-Sellers, B. (1996). Object-Oriented Metrics: Measures of Complexity, Prentice Hall.

Henderson-Sellers, B., Edwards, J. M. (1990). "The Object-Oriented Systems Life Cycle." Communications of the ACM **33**(9): 142-159.

Henderson-Sellers, B., Edwards, J. M. (1994). BOOKTWO of Object-Oriented Knowledge: The Working Object, Prentice Hall.

Heuer, A. (1992). Objekorientierte Datenbanken: Konzepte, Modelle, Systeme, Addison Wesley.

Hong, S., Goor, G. v. d., et al. (1992). A Comparison of Object-oriented Analysis and Design Methods. 26th Hawaiian international conference on System Sciences, IEEE Computer Science Press.

HOOD, T. G. (1990). Hood Reference Manual. Technical Report HOOD90.

Hörner, C., Heuer, A. (1991). EXTREM-The structural part of an object-oriented database model. Technical Report 91/5. Clausthal, Computer Science Department University of Clausthal.

Housel, B. C., Waddle, V., et al. (1979). The Functional Dependency Model for Logical Database Design. 5th International Conference on Very Large Databases, Rio de Janeiro.

Humphrey, W. S. (1995). A discipline for software engineering, Addison Wesley.

Ichbiah, J. D., Ed. (1980). Reference Maual for the Ada Programming Language: Proposed Standard document. Lecture Notes in Computer Science. New York, Springer.

IEEE (1989). Standard for a Software Quality Metrics Methodology. Technical Report P-1061/D20. New York, IEEE.

Ingalls, D. H. (1976). The Smalltalk-76 Programming System: Design and Implementation. Fifth Annual ACM Symposium on Programming Languages, Tucson, Arizona.

ISO (1990). Software Product Evaluation - Quality Characteristics and Guidelines for their Use. Technical Report ISO/IEC DIS 9126. New York, ISO.

Jacobson, I., et. al. (1992). Object-oriented Software Engineering - A use-case driven approach, ACM-Press/Addison Wesley.

Johnson, R. E., Opdyke, W. F. (1993). Refactoring and aggregation. International Symposium on Object Techniques for Advanced Software.

Kilov, H., Ross, J. (1994). Information Modelling: An Object-Oriented Approach, Prentice Hall.

Kim, H.-J. (1991). Algorithmic and Computational Aspects of Object-Oriented Schema Design. Object-Orinted Databases with Applications to CASE, Networks and VLSI CAD. R. Gupta and E. Horowitz, Prentice-Hall.

Kim, W. (1992). On Unifying Relational and Object-Oriented Database Systems. European Conf. on Object-Oriented Programming (ECOOP), Utrecht, Netherlands.

Kitchenham, B., Pfleeger, S. L., et al. (1995). "Towards a Framework for Software Measurement Validation." IEEE Transactions on Software Engineering **21**(12): 929-944.

Kitchenham, B., Pfleeger, S. L., et al. (1996). "Software Quality: The elusive target." IEEE Software **13**(1): 12-21.

Koenig, A. (1996). "Patterns and Antipatterns." Journal of Object-oriented programming **8**(1): 46-48.

Kruchten, P. B. (1995). "The 4+1 View Model of Architecture." IEEE Software **12**(6): 42-50.

Ku, C. S. (1991). An Object-Oriented Entity-Relationship Model. Int'l Conference on Computer Applications in Design, Simulation and Analysis, Las Vegas, NE.

Lausen, G., Vossen, G. (1996). Objekt-orienterte Datanbanken: modelle und Sprachen. München, Oldenbourg.

Lazimi, R. (1989). EER Model and Object-Oriented Representation for Data Management, ProcessModeling, and Decision Support. 8th Int'l Conference on the Entity-Relationship Approach.

Lieberherr, K. J. (1992). The Demeter Method for Object-oriented Software Engineering. Technical Report Demeter, Northeastern University.

Lieberherr, K. J., Holland, I. (1989). "Assuring Good Style for Object-Oriented Programs." IEEE Software **September**: 38-44.

Liskov, B. (1987). Data Abstraction and Hierarchy. OOPSLA 87 Addendum to the Proceedings.

Loomis, M. E. S., Shaw, A. V., et al. (1987). An Object Modelling Technique for Conceptual Design. European Conference on Object-Oriented Programming, Paris, France, Springer.

Lorenz, M., Kidd, J. (1994). Object-Oriented Software Metrics. New York, Prentice Hall.

Maffeis, S., Toenniessen, F., et al. (1999). Erfahrungen mit Java, dpunkt.

Maier, D. (1983). The Therory of Relational Databases, Computer Science Press, Rockville, MD.

Martin, J., Odell, J. J. (1992). Object-Oriented Analysis and Design. Englewood Cliffs, NJ., Prentice Hall.

Martin, R. C. (1995). "Object-Oriented Design Quality Metrics: An Analysis of Dependencys." Report on Object Analysis&Design **2**(3).

Martin, R. C. (1995a). Designing Object Oriented C++ Applications using the Booch Method, Prentice Hall.

Meyer, B. (1988). Object-Oriented Software Construction, Prentice Hall, Englewood Cliffs, NJ.

Nachouki, J., Chastang, M. P. (1991). From Entity-Relationship Diagram to An Object Oriented Database. 10th Int'l Conference on the Entity-Relationship Approach.

Navathe, S. B. (1989). OOER: Toward Making the E-R Approach Object-Oriented. 9th Int'l Conference on the Entity-Relationship Approach.

OMG (1997a). CORBAservices Specification. Technical Report 97-12-02. Framingham, Object Management Group.

OMG (1997b). UML Semantics. Technical Report ad/97-08-04, Rational Software.

Opdyke, W. F. (1992). Refactoring Object-Oriented Frameworks. Technical Report UIUC-CS-92-1759. Urbana, Illinois, Universty of Illinois.

Orfali, R., Harkey, D. (1997). Client/Server Programming with Java and CORBA, John Whiley&Sons.

Porter, A., Votta, L. (1997). "What makes inspections work?" IEEE Software November, December 14(6): 99-102.

Riel, A. (1995). Design Heuristics and their Relationship to Patterns. OOPSLA 1995, Austin, Texas.

Riel, A. (1996). Object-oriented Design Heuristics, Addison Wesley.

Rishe, N. (1988). Database design fundamentals, Prentice Hall.

Roberts, F. S. (1979). Measurement Theory with Applications to Decision-making, Utility, and the Social Sciences, Addison Wesley.

Royce, W. W. (1970). Managing the development of Large Software Systems: Concept and techniques. WesCon, Los Angeles.

Rumbaugh, J. (1995). "What is a method?" Journal of object-oriented programming 8(6): 10-16.

Rumbaugh, J., et. al. (1991). Object-Oriented Modeling and Design. Englewood Cliffs, NJ, Prentice Hall.

Sefika, M., Sane, A., et al. (1996). Monitoring Compliance of a Software System With Its High-Level Design Models. ICSE 96, Berlin, Germany.

Shaw, M. (1995). Patterns for Software Architectures. Pattern Languages of Program Design. J. Coplien and D. C. Schmidt, Addison Wesley.

Shlaer, S., Mellor, S. J., et al. (1988). The object-oriented method for analysis. 10th Structured Development Forum, San Francisco, CA.

SoftWired (1999). iBus Programmer's Manual. Technical Report Version 2.0. Zurich, SoftWired Inc.

Spaccapietra, S., Parent, C. (1992). ERC+: an object based entity relationship approach. Conceptual Modelling, Databases and CASE: An Integrated View of Information Systems Development. P. Loucopoulos and R. Zicari, Wiley.

Stonebraker, M., Moore, D. (1996). Object-Relational DBMSs - The next great wave. San Francisco, CA, Morgan Kaufmann.

Sutcliffe, A. G., Maiden, N. A. M. (1994). Domain Modeling for Reuse. 3rd international Conference on Software Reuse, IEEE Computer Society Press.

Teorey, T. J., Yang, D., et al. (1986). "A logical design methodology for relational databases using the extended entity-relationship model." ACM Computing Surveys, 18(2).

Tervonen, I. (1996). "Support for Quality-Based Design and Inspection." IEEE Software 13(1): 44-54.

TIBCO (1999a). TIB/Rendezvous. Technical Report TIB1, TIBCO Software Inc.

TIBCO (1999b). TIB/Object Bus. Technical Report TIB2, TIBCO Software Inc.

Triola, M. F. (1999). Elementary Statistics, Addison Wesley.

Vlissides, J., et al. (1996). Pattern Languages of Program Design -- 2. Reading, MA, Addison Wesley.

Weiser, M. D. (1982). "Programmers Use Slices when Debugging." Communications of the ACM 25(7): 446-452.

Wirfs-Brock, R., Wilkerson, B., et al. (1990). Designing Object-Oriented Software, Prentice-Hall, Englewood Cliffs, NJ.

Wirth, N. (1977). "Modula: A language for Modular programming." Software - Practice &Experience 7(1): 3-35.

Yao, S. B. (1985). Principles of Database Design, Vol. 1, Prentice-Hall, Englewood Cliffs, NJ.

Yourdon, E., Constantine, L. L. (1997). Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Englewood Cliffs, Yourdon Press/Prentice Hall.

Zuse, H. (1995). History of Software Measurement. Technical Report http://irb.cs.tu-berlin.de/~zuse/metrics/3-hist.html.

Zuse, H., Bollmann, P. (1989). "Using Measurement Theory to Describe the Properties and Scales of Static Software Complexity Metrics." SIGPLAN Notices 24(8): 23-33.

Zuse, H., Bollmann, P. (1992). Measurement Theory and Software Measures. BCS-FACS Workshop on Formal Aspects of Measurement, South Bank University, London, Springer.

Thomas Grotehen wurde am 5.12.1966 in Holzminden geboren. Er studierte von 1988-1993 Informatik an der technischen Universität Clausthal. Danach war er von 1993-1995 Senior Analytiker beim Schweizerischen Bankverein in Basel, von 1995-1998 Research Engineer bei der SYSTOR AG in Basel und von 1998-2001 Chefarchitekt bei der CustomerCare AG in Rheinfelden. Parallel dazu war er von 1993-2000 wissenschaftlicher Mitarbeiter der Datenbankgruppe Universität Zürich, wo er diese Dissertation angefertigt hat.